
# PCI-to-PCI Bridge Architecture Specification

**Revision 1.1**

**December 18, 1998**

**Revision History**

| Revision | Issue Date | Comments |
|---|---|---|
| 1.0 | 4/5/94 | Original issue |
| 1.1 | 12/18/98 | Update to include target initial latency requirements. |

The PCI Special Interest Group disclaims all warranties and liability for the use of this document and the information contained herein and assumes no responsibility for any errors that may appear in this document, nor does the PCI Special Interest Group make a commitment to update the information contained herein.

Contact the PCI Special Interest Group office to obtain the latest revision of the specification.

Questions regarding the PCI specification or membership in the PCI Special Interest Group may be forwarded to:

PCI Special Interest Group
2575 N.E. Kathryn #17
Hillsboro, Oregon 97124
1-800-433-5177 (USA)
503-693-6232 (International)
503-693-8344 (Fax)
pcisig@pcisig.com
http://www.pcisig.com

DISCLAIMER

This PCI-to-PCI Bridge Architecture Specification is provided "as is" with no warranties whatsoever, including any warranty of merchantability, noninfringement, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. The PCI SIG disclaims all liability for infringement of proprietary rights, relating to use of information in this specification. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted herein.

Intel and Pentium are registered trademarks of Intel Corporation.

All other product names are trademarks, registered trademarks, or servicemarks of their respective owners.

# CONTENTS

## CHAPTER 1 INTRODUCTION



## CHAPTER 2 BRIDGE REQUIREMENTS



## CHAPTER 3 CONFIGURATION

## CHAPTER 4 ADDRESS DECODING



## CHAPTER 5 BUFFER MANAGEMENT

## CHAPTER 12 VGA SUPPORT



## CHAPTER 13 SLOT NUMBERING

# Preface

## Scope

This specification defines the behavior of a compliant PCI-to-PCI bridge. A PCI-to-PCI bridge that conforms to this specification and the *PCI Local Bus Specification* is a compliant implementation. Compliant bridges may differ from each other in performance and to some extent functionality.

## Related Documents

This specification assumes that the reader has a working knowledge of the *PCI Local Bus Specification* and is familiar with other PCI specifications. Refer to the PCI SIG web page for the latest list of specifications and revision levels.

Following publication of the *PCI-to-PCI Bridge Architecture Specification*, there may be future approved errata and/or approved changes to the specification prior to the issuance of another formal revision. To assure designs meet the latest level requirements, designers of PCI-to-PCI bridges must refer to the PCI SIG home page at http://www.pcisig.com, in the members-only section, for any approved changes.

# Chapter 1
# Introduction

## 1.1. Goals and Non-Goals of this Specification

This specification establishes the requirements that a PCI-to-PCI bridge must meet to be compliant to this specification and the *PCI Local Bus Specification*. In addition, the requirements for optional extensions are specified. This specification does not describe the implementation details of any particular requirement or optional feature of a PCI-to-PCI bridge, nor is it a goal of this specification to describe any particular PCI-to-PCI bridge implementation. However, some recommendations are provided for *some* implementation-specific features that can be provided by a PCI-to-PCI bridge.

## 1.2. Overview and Terminology

A PCI-to-PCI bridge provides a connection path between two independent PCI buses. The primary function of the bridge is to allow transactions to occur between a master on one PCI bus and a target on the other PCI bus. PCI-to-PCI bridges provide system and expansion board designers the ability to overcome electrical loading limits by creating hierarchical PCI buses. To aid in the discussion of PCI-to-PCI bridge architecture, the following terminology is used in this document:

*bridge* - In this document, the word *bridge* when used by itself is equivalent to the term *PCI-to-PCI bridge.* Other types of bridges such as expansion bus bridges or host bus bridges are always explicitly named.

*downstream* - Transactions that are forwarded from the primary interface to secondary interface of a bridge are said to be flowing downstream.

*originating bus* - The master of a transaction that crosses a bridge is said to reside on the originating bus.

*primary interface* - The PCI interface of the bridge that is connected to the PCI bus closest to the CPU is referred to as the primary PCI interface.

*secondary interface* - The PCI interface of the bridge that is connected to the PCI bus farthest from the CPU is referred to as the secondary PCI interface.

*destination bus* - The target of a transaction that crosses a bridge is said to reside on the *destination* bus.

*upstream* - Transactions that are forwarded from the secondary interface to primary interface of a bridge are said to be flowing upstream.

Thus, a bridge has two PCI interfaces, the primary and secondary. Each interface is capable of both master and target operation. The bridge functions as a target on the *originating* bus on behalf of the target that actually resides on the *destination* bus. Likewise, the bridge functions as a master on the *destination* bus on behalf of the originating master that actually resides on the *originating* bus.

Figure 1-1 illustrates two typical applications for a bridge. The first application is the use of a bridge to create a second PCI bus segment to which additional PCI connectors are added. This bus segment is labeled in the figure as PCI Bus 1. In this example, the primary interface of bridge 1 is connected to PCI bus 0 while its secondary interface is connected to PCI bus 1. The second application example is the use of a bridge to create a PCI bus segment on an expansion board that allows multiple PCI devices to reside on a single expansion board. In this example, the primary interface of bridge 2 is connected to PCI bus 1 and its secondary interface is connected to PCI bus 2. Note that the number assigned to the bridge corresponds to the number of the bus segment spawned by the bridge. In this example, the host bridge is considered to be bridge number 0 and spawns PCI bus segment 0.

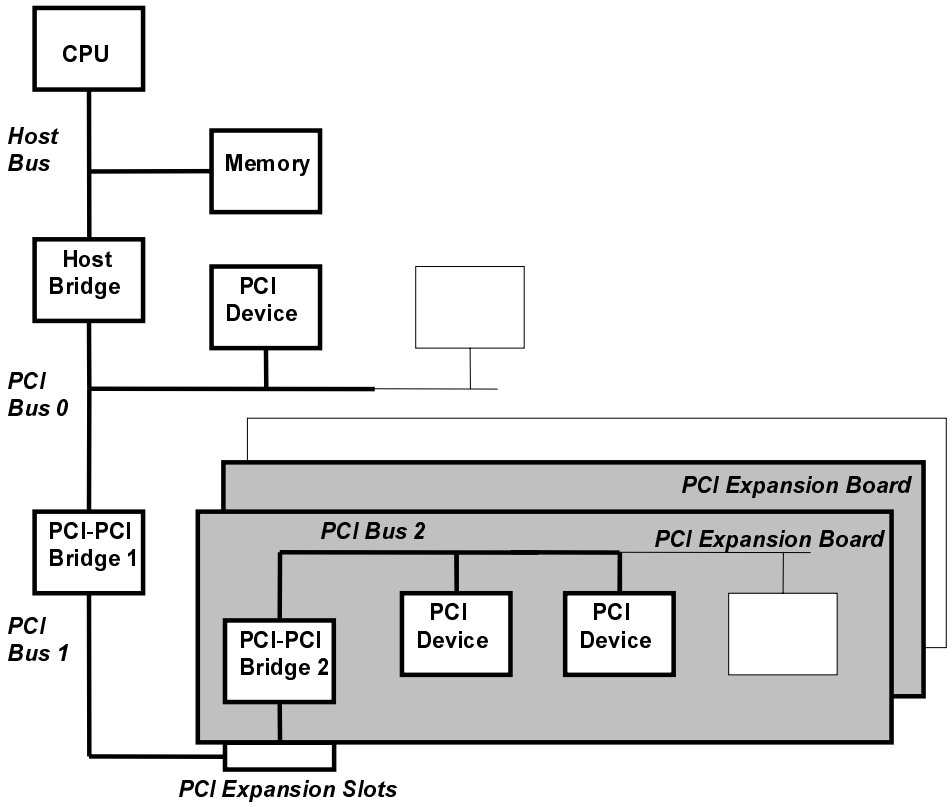



**Figure 1-1: Typical Bridge Applications**

A bridge allows transactions between a master on one PCI interface and a target on the other interface as illustrated in Figure 1-2. The target interface on one bus is connected to the master interface on the other bus. The blocks between the data path of the primary and secondary interfaces provide any necessary transaction address and data buffering. The target block connected to the primary PCI interface must support PCI configuration space. The bridge basically consists of four state machines—two masters and two targets. Each of the master and target interface state machines must adhere to the requirements of the *PCI Local Bus Specification*.

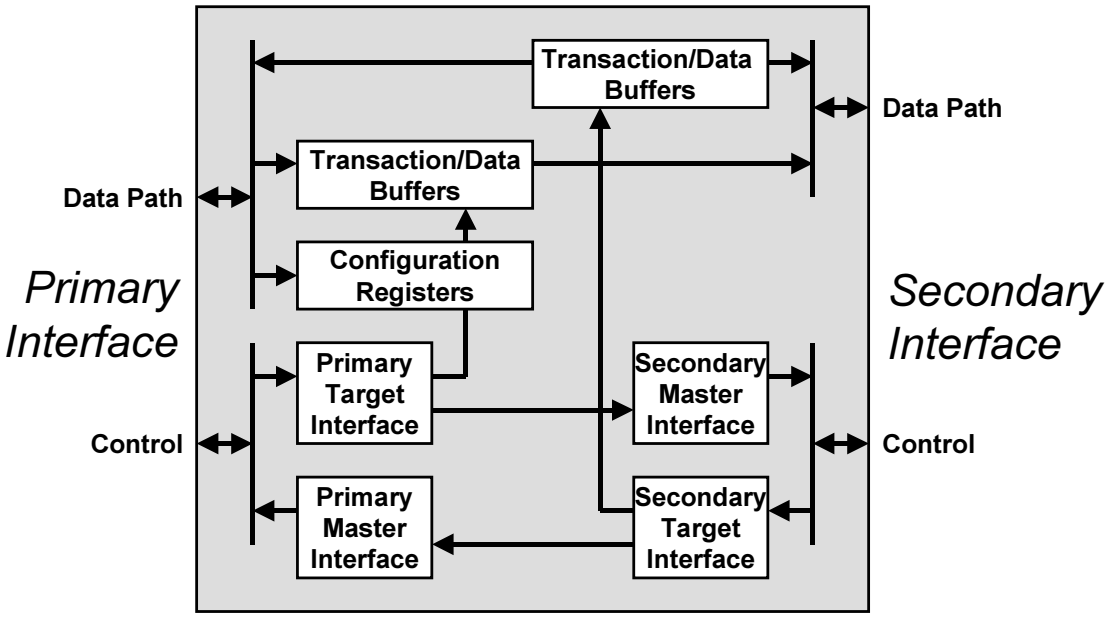



**Figure 1-2: Example Bridge Block Diagram**

# Chapter 2
# Bridge Requirements

## 2.1. Summary of Key Requirements

A summary of key bridge requirements are listed below:

- A bridge must be compliant with the current *PCI Local Bus Specification*. This includes the following requirements:
  - The bridge must adhere to the electrical loading limits for all PCI signals. When a bridge is used on the expansion board, the bridge is limited to a single connection per PCI signal (for example, **CLK**). As a result, when the secondary PCI bus runs synchronously to the primary PCI bus, the bridge must buffer the **CLK** signal received from the expansion board connector for distribution to other PCI devices connected to the secondary bus. See Section 10.2.1. for additional clock buffering considerations.
  - A bridge must support the range of operation from DC to 33 MHz. A bridge may optionally support 66 MHz operation as defined by the *PCI Local Bus Specification*. The relationship between the primary interface and secondary interface clocks of a bridge is implementation specific.
  - The PCI connector does not support side-band signals. Therefore, a bridge cannot require any side-band signals for correct operation when used in expansion board applications. A bridge must maintain transaction ordering as described in Appendix E of the *PCI Local Bus Specification* when transactions cross the bridge in either direction.
  - A bridge must adhere to the 16-clock target initial latency and 8-clock target subsequent latency rules for all transactions, including those that are forwarded across the bridge as well as those that access registers internal to the bridge. The *PCI Local Bus Specification* grants exceptions to the target initial latency during initialization time.
- A bridge must comply with the requirements set forth in the remainder of this document. This includes the required capabilities listed below:
  - Configuration register space adhering to the PCI-to-PCI bridge Type 1 Header format
  - Hierarchical configuration transaction support
  - Memory mapped I/O address space for transaction forwarding

  - Posting of memory write transactions
  - Support of Delayed Transactions (for non-posted transactions)
  - Support the forwarding of DAC upstream
- If a bridge provides the arbiter for the secondary bus, it must be designed to prevent deadlocks. The bridge is required to implement a fairness algorithm to avoid potential deadlocks. Refer to Section 8.2. for more details.

## 2.2. Capabilities Not Supported

Listed below are capabilities that are precluded by this specification. There may be other capabilities precluded by this specification that do not appear on the list below.

- Support for unusual configurations (some examples are listed below).
  - Using two bridges to connect to a common secondary bus and different primary buses
  - Multiple bridges connecting to same primary bus and same secondary bus
  - Two bridges where the primary interface of one bridge is connected to the secondary interface of the other and vice versa
- Forwarding of Special Cycle transactions. Special Cycle transactions are only supported through Configuration Type 1 transactions.
- Forwarding of Interrupt Acknowledge transactions.

Listed below are capabilities that are not controlled by this specification. It *may be possible* for a bridge to provide support for these (and other) capabilities, but this specification does not attempt to make provisions for their support.

- Support of downstream devices that require mapping to the first 1 MB of memory space.
- Support of downstream bridges to non-PCI buses.
- ISA compatibility addressing for devices other than VGA.
- Access by ISA masters or ISA DMA of devices located on hierarchical PCI buses. It is assumed that ISA masters or ISA DMA access system memory only.
- Primary boot ROM on secondary interface. The bridge must be configured before access of downstream devices can occur.

## 2.3. Optional Capabilities

Listed below are capabilities that a bridge is not required to support but for which provisions have been made by this specification. These may be optionally supported by a bridge provided they adhere to the requirements and guidelines established in this specification and the *PCI Local Bus Specification*.

- Support for optional address ranges:
  - I/O address range
  - Prefetchable memory address range
- VGA support:
  - VGA addressing
  - VGA palette snooping
- JTAG
- 64-bit addressing on the primary interface
- 64-bit data path
- Arbitration support for secondary bus devices
- Expansion ROM
- Subtractive decoding

# Chapter 3
# Configuration

## 3.1. Overview of Hierarchical Configuration

The *PCI Local Bus Specification* defines two configuration transaction types, Type 0 and Type 1, which are illustrated in Figure 3-1. The two configuration address formats are distinguished by the value of address bits **AD[1::0]**. A Type 0 configuration transaction is used to access a device on the current bus segment and a Type 1 configuration transaction is used to access a device that resides behind a bridge.

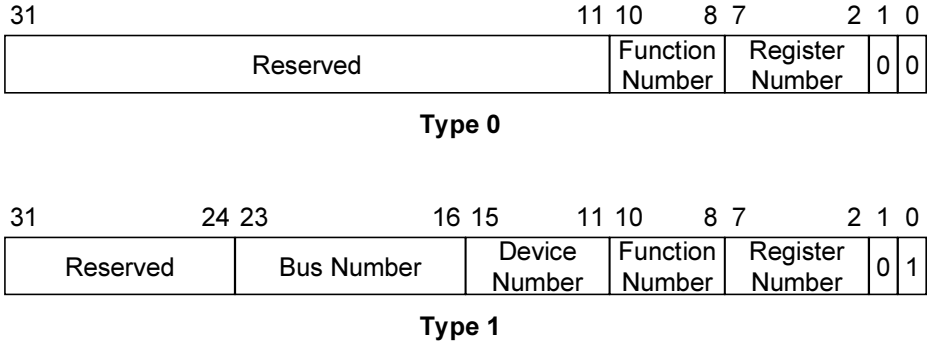



**Figure 3-1: Configuration Type 0 and Type 1 Address Format**

If address bits **AD[1::0]** are 00b during a configuration transaction, then a Type 0 configuration transaction is being used. A Type 0 configuration transaction is not forwarded across a bridge but is used to configure a bridge or other PCI devices that are connected to the PCI bus on which the Type 0 configuration transaction is generated.

If address bits **AD[1::0]** are 01b during a configuration transaction, then a Type 1 configuration transaction is being used. A Type 1 configuration transaction is used to address a device that does not reside on the current bus segment and may be forwarded to another bus segment by a bridge.

The following sections describe the support provided by bridges for Type 0 and Type 1 configuration transactions.

### 3.1.1. Type 0 Configuration Transaction Support

A bridge only responds to Type 0 configuration transactions on its primary PCI interface when being configured. A bridge ignores Type 0 configuration transactions that originate on the secondary interface of the bridge. Thus, the bridge does not implement **IDSEL** on its secondary interface. A Type 0 configuration transaction is used to configure the bridge and is not forwarded downstream by the bridge (from its primary to secondary interface).

PCI devices, including bridges, are selected by a PCI configuration transaction when the following conditions are all true:

- **IDSEL** is asserted;
- The PCI bus command is a Configuration Read or Configuration Write;
- Address bits **AD[1::0]** are 00 (during the address phase of the transaction); and
- If a multifunction device, address bits **AD[10:08]** select an implemented function.

During a configuration transaction, address bits **AD[7::2]** select a DWORD (longword) register in the device's 256-byte configuration address space. Address bits **AD[31::11]** are ignored by devices during configuration transactions.

### 3.1.2. Type 1 Configuration Transaction Support

During a Type 1 configuration transaction, address bits **AD[23::16]** specify a unique PCI bus in the PCI hierarchy on which the target of the transaction resides. The bridge compares the specified bus number with three configuration registers that are programmed by initialization code to determine whether to claim and forward a Type 1 configuration transaction across the bridge. The three configuration registers are listed below.

- Primary Bus Number (configuration register offset 18h)
- Secondary Bus Number (configuration register offset 19h)
- Subordinate Bus Number (configuration register offset 1Ah)

If the bridge claims a Type 1 configuration transaction, these three registers also determine how the transaction is forwarded across the bridge. The following sections discuss the Type 1 transaction forwarding options available to the bridge for both the primary and secondary interfaces.

#### 3.1.2.1. Primary Interface

If a Type 1 configuration transaction occurs on the primary interface of the bridge, the bridge either ignores the transaction or claims the transaction and forwards it to its secondary interface, as specified below.

The bridge ignores a Type 1 configuration transaction on its primary interface, if the bus number specified by address bits **AD[23::16]** does not fall within the range of bus numbers specified by the Secondary Bus Number (inclusive) and Subordinate Bus Number (inclusive) registers. In

this case, the Type 1 configuration transaction is specifying a bus number that is *not* located behind the bridge.

The bridge claims a Type 1 configuration transaction on its primary interface, if the bus number specified by address bits **AD[23::16]** falls within the range of bus numbers specified by the Secondary Bus Number (inclusive) and Subordinate Bus Number (inclusive) registers. In this case, the Type 1 configuration transaction is specifying a bus number that *is* located behind the bridge. If a bridge forwards a Type 1 configuration transaction to its secondary interface, the bridge must use one of the following methods:

- Convert the transaction to a Type 0 configuration transaction (to access the configuration registers of a device attached to the secondary interface of the bridge); or
- Forward the transaction unmodified (as a Type 1 configuration transaction to access a device that does not reside on the secondary interface of the bridge but is located on a bus segment further downstream); or
- Convert the transaction to a Special Cycle transaction.

The following sections describe when each of these Type 1 configuration transaction forwarding methods is used.

## 3.1.2.1.1. Type 1 to Type 0 Conversion

If a Type 1 configuration transaction occurs on the primary interface of the bridge and the bus number specified by address bits **AD[23::16]** matches the Secondary Bus Number, the bridge claims the transaction, converts it to a Type 0 transaction (as described below), and forwards the Type 0 transaction to its secondary interface. In this case, a device connected to the secondary interface of the bridge is the target of the resulting Type 0 configuration transaction.

To convert the forwarded transaction from a Type 1 to a Type 0 configuration transaction, the bridge must do the following:

- Modify address bits **AD[1::0]** so that they are 00b.
- Decode the device number field specified by address bits **AD[15::11]** of the Type 1 transaction to select the pattern as specified by Table 3-1 to drive on address bits **AD[31::16]** during the resulting Type 0 transaction on the secondary bus.

Address bits **AD[10::2]** from the Type 1 configuration transaction on the primary interface must be passed unmodified by the bridge to the resulting Type 0 configuration transaction on its secondary interface. The value driven on address bits **AD[15::11]** by the bridge during the resulting Type 0 configuration transaction is not specified.

Table 3-1 specifies the pattern to be driven by the bridge on address bits **AD[31::16]** for each encoding of the device number field (address bits **AD[15::11]**). If **AD[15]** is 0, the pattern driven on address bits **AD[31::16]** during the resulting Type 0 transaction on the secondary bus will have one (and only one) bit set to a 1 (all other bits will be zero). This allows board designers to use address bits **AD[31::16]** as **IDSEL** signals for the devices attached to the secondary interface of the bridge (by connecting a unique bit of **AD[31::16]** to each of the **IDSEL** pins of the devices attached to the secondary bus). If **AD[15]** is 1, the bridge must drive

address bits **AD[31::16]** to 0 (all bits) during the resulting Type 0 transaction on the secondary bus.

When doing a Type 1 to Type 0 conversion, a bridge is permitted to provide additional methods of **IDSEL** generation but must always convert **AD[31::16]** as specified by Table 3-1. Board designers may use alternate methods of **IDSEL** generation that are independent from those provided by the bridge. However, the board designer must follow the interrupt routing requirements specified in Chapter 9.

**Table 3-1: IDSEL Generation**

| Primary Address AD[15::11] | Secondary Address AD[31::16] |
|---|---|
| 00000 | 0000 0000 0000 0001 |
| 00001 | 0000 0000 0000 0010 |
| 00010 | 0000 0000 0000 0100 |
| 00011 | 0000 0000 0000 1000 |
| 00100 | 0000 0000 0001 0000 |
| 00101 | 0000 0000 0010 0000 |
| 00110 | 0000 0000 0100 0000 |
| 00111 | 0000 0000 1000 0000 |
| 01000 | 0000 0001 0000 0000 |
| 01001 | 0000 0010 0000 0000 |
| 01010 | 0000 0100 0000 0000 |
| 01011 | 0000 1000 0000 0000 |
| 01100 | 0001 0000 0000 0000 |
| 01101 | 0010 0000 0000 0000 |
| 01110 | 0100 0000 0000 0000 |
| 01111 | 1000 0000 0000 0000 |
| 1xxxx | 0000 0000 0000 0000 |

### 3.1.2.1.2. Type 1 to Type 1 Forwarding

If a Type 1 configuration transaction occurs on the primary interface of the bridge and the bus number specified by address bits **AD[23::16]** is within the range of bus numbers between the Secondary Bus Number (exclusive) and the Subordinate Bus Number (inclusive), the bridge claims the transaction and forwards it unmodified to its secondary interface. In this case, the target of the configuration transaction does not reside on the secondary interface of the bridge but is located on a bus segment further downstream. The Type 1 configuration transaction generated on the secondary bus is potentially addressing a device that resides behind other bridges that may be attached to this bridge's secondary interface. Note that a bridge uses exactly the same address, bus command, byte enables, and data (if a write) received on the primary interface to generate the Type 1 transaction on its secondary interface (there is no conversion).

### 3.1.2.1.3. Type 1 to Special Cycle Conversion

A bridge claims a Type 1 configuration write transaction that occurs on its primary interface and converts it to a Special Cycle on its secondary interface when the following conditions are met:

- The bus number specified by address bits **AD[23::16]** matches the Secondary Bus Number of the bridge;
- The device number specified by address bits **AD[15::11]** is all ones (equals 11111b);
- The function number specified by address bits **AD[10::8]** is all ones (equals 111b); and
- The register number specified by address bits **AD[7::2]** is all zeros (equals 000000b).

The address during a Special Cycle is ignored by PCI devices, and the bridge is allowed to drive any value on **AD[31::00]** during the address phase. The data for the Special Cycle on the secondary interface is the write data received from the Type 1 configuration transaction on the primary interface. Note that Type 1 configuration transactions which specify conversion (by a bridge) to a Special Cycle are restricted to a data burst length of 1 (see the *PCI Local Bus Specification* for more information).

### 3.1.2.2. Secondary Interface

Support of configuration transactions on the secondary interface of a bridge is limited as compared to support on the primary interface. The bridge is only allowed to claim a Type 1 configuration write transaction on its secondary interface that specifies a conversion to a Special Cycle on a bus segment that resides above the bridge. In all other conditions, the bridge is required to ignore a configuration transaction on its secondary interface. This means the bridge ignores the following configuration transactions on its secondary interface:

- Type 0 configuration transactions (read or write);
- Type 1 configuration read transactions; or

- Type 1 configuration write transactions if the bus number specified by address bits **AD[23::16]** is in the range of bus numbers between the Secondary Bus Number (inclusive) and Subordinate Bus Number (inclusive) of the bridge
- Type 1 configuration write transaction that does not accurately specify conversion to a Special Cycle

### 3.1.2.2.1. Type 1 to Type 1 Forwarding

A bridge will forward a Type 1 configuration write transaction unmodified to its primary interface, provided the following conditions are met:

- The device number specified by address bits **AD[15::11]** is all ones (equals 11111b);
- The function number specified by address bits **AD[10::8]** is all ones (equals 111b);
- The register number specified by address bits **AD[7::2]** is zero (equals 000000b); and
- The bus number specified by address bits **AD[23::16]** is not between the Secondary Bus Number (inclusive) and the Subordinate Bus Number (inclusive).

In this case, the bridge generates a Type 1 configuration write transaction on the primary interface using exactly the same address and data information that was contained in the transaction on the secondary interface. The Type 1 transaction generated on the primary interface of the bridge can then be claimed by another bridge and converted to a Special Cycle transaction on the destination bus segment.

### 3.1.2.2.2. Type 1 to Special Cycle Conversion

A bridge will convert a Type 1 configuration write transaction received on its secondary interface to a Special Cycle on its primary interface provided the following conditions are met:

- The device number specified by address bits **AD[15::11]** is all ones (equals 11111b);
- The function number specified by address bits **AD[10::8]** is all ones (equals 111b);
- The register number specified by address bits **AD[7::2]** is zero (equals 000000b); and
- The bus number specified by address bits **AD[23::16]** matches the Primary Bus Number of the bridge.

The address during a Special Cycle is ignored by PCI devices and the bridge is allowed to drive any value on **AD[31::00]** during the address phase of the Special Cycle transaction. The data for the Special Cycle on the primary interface will be the write data from the Type 1 configuration transaction on the secondary interface. Type 1 configuration transactions that specify conversion (by a bridge) to a Special Cycle are restricted to a burst length of 1 (see the *PCI Local Bus Specification* for more information).

## 3.2. PCI-to-PCI Bridge Configuration Space Header Format

The *PCI Local Bus Specification* requires all devices, including a PCI-to-PCI bridge, to implement a 256-byte configuration register address space. The first 64 bytes in each device's PCI Configuration Space must adhere to a standard configuration header format. The remaining 192 bytes of the Configuration Space may be used for additional capabilities as defined by the Capabilities Pointer or for device-specific purposes.

The 64-byte header format for a bridge is defined in Figure 3-2. The first 16 bytes of the bridge header implement the common format for all devices as required by the *PCI Local Bus Specification*. The next 48 bytes of the device's Configuration Space are Header Type specific. A Header Type value of 1 indicates that the device follows the bridge register layout, which is defined in this specification. The following sections define the basic behavior of configuration registers and how reset affects them. Then a brief review of the common registers is presented, followed by a detailed specification of the bridge specific registers.

| 31 24 | 23 16 | 15 8 | 7 0 | |
|---|---|---|---|---|
| **Device ID** | | **Vendor ID** | | **00 h** |
| **Status** | | **Command** | | **04h** |
| **Class Code** | | | **Revision ID** | **08h** |
| **BIST** | **Header Type** | **Primary Latency Timer** | **Cacheline Size** | **0Ch** |
| **Base Address Register 0** | | | | **10h** |
| **Base Address Register 1** | | | | **14h** |
| **Secondary Latency Timer** | **Subordinate Bus Number** | **Secondary Bus Number** | **Primary Bus Number** | **18h** |
| **Secondary Status** | | **I/O Limit** | **I/O Base** | **1Ch** |
| **Memory Limit** | | **Memory Base** | | **20h** |
| **Prefetchable Memory Limit** | | **Prefetchable Memory Base** | | **24h** |
| **Prefetchable Base Upper 32 Bits** | | | | **28h** |
| **Prefetchable Limit Upper 32 Bits** | | | | **2Ch** |
| **I/O Limit Upper 16 Bits** | | **I/O Base Upper 16 Bits** | | **30h** |
| **Reserved** | | | **Capabilities Pointer** | **34h** |
| **Expansion ROM Base Address** | | | | **38h** |
| **Bridge Control** | | **Interrupt Pin** | **Interrupt Line** | **3Ch** |

**Figure 3-2: PCI-to-PCI Bridge Configuration Registers**

### 3.2.1. Access of Reserved Registers

Read accesses to reserved or optional registers which are not implemented must complete normally and return a data value of zero when read. Writes to reserved registers must be treated as no-ops unless otherwise specified by this document. That is, the write access must be completed normally and the write data has no effect and is simply discarded.

### 3.2.2. Access of Reserved Bit Fields

Software must be careful when accessing registers that have bit fields reserved for future use. For read accesses, software must use appropriate masks to extract the defined bits and may not rely on reserved bits being any particular value. For write accesses, software must ensure that the values of reserved bit positions are preserved. That is, the values of the reserved bit positions must first be read, merged with the new values for other bit positions, and the merged data then written back.

### 3.2.3. Reset Events

The assertion of **RST#** on the primary interface affects the state of some bits in the bridge configuration registers. The reset state of each bit is described in the bit definitions of each register when applicable. The assertion of **RST#** on the secondary interface does not affect the state of any register bits in the standard portion of the bridge configuration header (the first 64 bytes). However, setting the Secondary Bus Reset bit (bit 6) in the Bridge Control register (refer to Section 3.2.5.17.) does affect the internal state of the bridge.

### 3.2.4. Common Format Configuration Registers

The following sections will give a brief description of the common format registers that all PCI devices must support. For a more detailed discussion, refer to the *PCI Local Bus Specification*. All registers and bits are required unless explicitly specified to be optional.

### 3.2.4.1. Vendor ID Register

The Vendor ID register identifies the manufacturer of the device and is assigned by the PCI Special Interest Group to insure uniqueness. The Vendor ID register must be implemented as a read-only register.

### 3.2.4.2. Device ID Register

The Device ID register identifies the particular device and is assigned by the vendor. The Device ID register must be implemented as a read-only register.

## 3.2.4.3. Command Register

The Command register controls how the bridge behaves on the primary interface and is the same as all devices with the exception of the VGA Palette Snoop bit. Because a bridge has two interfaces, some specific clarification of the Command register bits is needed and appears in Table 3-2. In most cases, the bits in this register affect the behavior of the bridge's primary interface only (exceptions are specified).

**Table 3-2: Command Register**

| Command Register Bit | Bit Function | PCI-to-PCI Bridge Specific Notes |
|---|---|---|
| 0 | I/O Space Enable | Controls the bridge's response as a target to I/O transactions on the primary interface that address a device that resides behind the bridge (see Section 3.2.5.6. I/O Base and Limit registers) or locations within the bridge itself (see Section 3.2.5.1.). If the bridge does not support an I/O address range or I/O mapped BAR, then this bit must be implemented as a read-only bit that returns 0 when read. If the bridge implements an I/O address range or I/O mapped BAR, then this bit must be implemented as a read/write bit. The default state of this bit after reset must be 0.<br><br>The state of internal transaction buffers is not specified when this bit is disabled after being enabled. The bridge can choose how it behaves when this condition occurs. Note: software cannot count on the bridge retaining state and resuming without loss of data when the bit is re-enabled.<br><br>0 - ignore I/O transactions on the primary interface<br><br>1 - enable response to I/O transactions on the primary interface |

| | | |
|---|---|---|
| 1 | Memory Space Enable | Controls the bridge's response as a target to memory accesses on the primary interface that address a device that resides behind the bridge in both the memory mapped I/O and prefetchable memory ranges (see Sections 3.2.5.8. and 3.2.5.9.) or targets a location within the bridge itself (see Section 3.2.5.1.). A bridge must implement this bit as a read/write bit (support of a memory address range is required). The default state of this bit after reset must be 0.<br><br>The state of internal transaction buffers is not specified when this bit is disabled after being enabled. The bridge can choose how it behaves when this condition occurs. Note: software cannot count on the bridge retaining state and resuming without loss of data when the bit is re-enabled.<br><br>0 - ignore all memory transactions on the primary interface<br><br>1 - enable response to memory transactions on the primary interface |
| 2 | Bus Master Enable | Controls the bridge's ability to operate as a master on the primary interface when forwarding memory or I/O transactions from the secondary interface to the primary interface on behalf of a master on the secondary interface. This bit does not affect the ability of a bridge to forward or convert configuration transactions from the secondary interface to the primary interface. Note that when this bit is zero, the bridge must disable response as a target to all memory or I/O transactions on the secondary interface (they cannot be forwarded to the primary interface). A bridge must implement this bit as a read/write bit. The default state of this bit after reset must be 0.<br><br>The state of internal transaction buffers is not specified when this bit is disabled after being enabled. The bridge can choose how it behaves when this condition occurs. Note: software cannot count on the bridge retaining state and resuming without loss of data when the bit is re-enabled.<br><br>0 - do not initiate memory or I/O transactions on the primary interface and disable response to memory and I/O transactions on secondary interface<br><br>1 - enable the bridge to operate as a master on the primary interface for memory and I/O transactions forwarded from the secondary interface |

| | | |
|---|---|---|
| 3 | Special Cycle Enable | A bridge does not respond as a target to Special Cycle transactions, so this bit is defined as read-only and must return 0 when read. |
| 4 | Memory Write and Invalidate Enable | A bridge that does not originate a Memory Write and Invalidate transaction (unless forwarding a transaction for another master) implements this bit as read-only with a value of 0. A bridge is allowed to convert a Memory Write transaction to a Memory Write and Invalidate transaction when conditions for Memory Write and Invalidate command usage are meet (see Section 5.2.1. for details). Bridges that implement such a feature must implement this bit as a read/write bit with a default state of 0 after reset. |
| 5 | VGA Palette Snoop Enable | Controls the bridge's response to VGA-compatible palette write accesses. The definition of this bit for a bridge is different than the *PCI Local Bus Specification* definition for devices with a type 0 configuration header.<br><br>Implementation of VGA palette snooping by a bridge is optional. If VGA palette snooping is not supported, this bit must be implemented as read-only with a value of 0. If a bridge supports VGA palette snooping, this bit must be implemented as a read/write bit with a default state of 0 after reset.<br><br>If this bit is set, I/O writes in the first 64 KB of the I/O address space (**AD[31::16]** is 0000h) with address bits **AD[9::0]** equal to 3C6h, 3C8h, and 3C9h (inclusive of ISA aliases - **AD[15::10]** are not decoded and may be any value) must be positively decoded on the primary interface and forwarded to the secondary interface. Conversely, these same addresses must be ignored by the bridge on the secondary interface.<br><br>0 - ignore VGA palette accesses on the primary interface<br><br>1 - enable positive decoding response to VGA palette writes on the primary interface with I/O address bits **AD[9::0]** equal to 3C6h, 3C8h, and 3C9h (inclusive of ISA aliases - **AD[15::10]** are not decoded and may be any value) |
| 6 | Parity Error Response | Controls the bridge's response to address and data parity errors on its primary interface. If this bit is set, the bridge must take its normal action when a parity error is detected. If this bit is cleared, the bridge must ignore any parity errors that it detects and continue normal operation. A bridge must implement this bit as a read/write bit with a default state of 0 after reset. |

| | | |
|---|---|---|
| 7 | Wait Cycle Control | Controls address/data stepping by the bridge (affects both interfaces). A bridge that never does stepping must hardwire this bit to 0. A bridge that always does stepping must hardwire this bit to 1. A bridge that can do either must make this a read/write bit with a default state of 1 after reset.<br>0 - address/data stepping is disabled<br>1 - address/data stepping is enabled |
| 8 | SERR# Enable | Controls the enable for the **SERR#** driver on the primary interface. A bridge must implement this bit as a read/write bit. The default state of this bit after reset must be 0.<br>0 - disable the **SERR#** driver on the primary interface<br>1 - enable the **SERR#** driver on the primary interface |
| 9 | Fast Back-to-Back Enable | Controls the ability of the bridge to initiate fast back-to-back transactions to different devices on the primary interface. A bridge that cannot initiate fast back-to-back transactions must implement this bit as read-only with a value of 0. A bridge that is capable of initiating fast back-to-back transactions must implement this bit as a read/write bit with a default state of 0 after reset. During system initialization, configuration software will set this bit if *all* devices on the primary interface are capable of fast back-to-back operation.<br>0 - disable the bridge from initiating fast back-to-back transactions on the primary interface<br>1 - enable the bridge to initiate fast back-to-back transactions on the primary interface |
| 15::10 | reserved | These bits are reserved for future use by the PCI SIG, are read-only, and must return zero when read. |

## 3.2.4.4. Status Register

The Status register provides information about the primary interface to the system. Table 3-3 provides the specific details for each bit as they apply to a bridge.

**Table 3-3: Status Register**

| Status Register Bit | Bit Function | PCI-to-PCI Bridge Specific Notes |
|---|---|---|
| 3::0 | reserved | These bits are reserved for future use by the PCI SIG and must be implemented as read-only bits, which return 0 when read. |
| 4 | Capabilities List | This bit indicates whether or not the bridge implements a Capabilities Pointer register pointing to a linked-list data structure of new capabilities. Support of the Capabilities List by a bridge is optional.<br>0 - the bridge does not support the Capabilities List<br>1 - the bridge supports the Capabilities List (Offset 34h is the pointer to the data structure) |
| 5 | 66 MHz Capable | This bit indicates whether or not the primary interface of the bridge is capable of operating at 66 MHz. Support of 66 MHz operation by a bridge is optional. This bit must be implemented as a read-only bit.<br>0 - the primary interface of the bridge is not capable of 66 MHz operation<br>1 - the primary interface of the bridge is capable of 66 MHz operation |
| 6 | reserved | This bit is reserved for future use and must be implemented as a read-only bit which returns 0 when read. |
| 7 | Fast Back-to-Back Capable | This bit indicates whether or not the primary interface of the bridge is capable of decoding fast back-to-back transactions when the transactions are from the same master but to different targets. (A bridge is required to support fast back-to-back transactions as a target from the same master.) This bit must be implemented as a read-only bit.<br>0 - the primary interface of the bridge is not capable of decoding fast back-to-back transactions to different targets<br>1 - the primary interface of the bridge is capable of decoding fast back-to-back transaction to different targets |

| | | |
|---|---|---|
| 8 | Master Data Parity Error | This bit is used to report the detection of a parity error by the bridge when it is the master of the transaction. This bit is set if the following three conditions are all true:<br>• The bridge is the bus master of the transaction on the primary interface;<br>• The bridge asserted **PERR#** (read transaction) or detected **PERR#** asserted (write transaction); and<br>• The Parity Error Response bit in the Command register is set.<br>Once set, this bit remains set until it is reset by writing a 1 to this bit location. A bridge must implement this bit and the default state is 0 after reset.<br>0 - no parity error detected on the primary interface<br>1 - parity error detected on the primary interface |
| 10::9 | **DEVSEL#** Timing | This read-only bit field encodes the timing of the primary interface **DEVSEL#** as listed below. The encoding must indicate the slowest response time that the bridge uses to assert **DEVSEL#** on its primary interface when it is responding as a target to any PCI transaction except[1] a Configuration Read or Configuration Write.<br>00 - fast **DEVSEL#** decoding<br>01 - medium **DEVSEL#** decoding<br>10 - slow **DEVSEL#** decoding<br>11 - reserved |
| 11 | Signaled Target-Abort | This bit reports the signaling of a Target-Abort termination by the bridge, when it responds as the target of a transaction on its primary interface. Once set, this bit remains set until it is reset by writing a 1 to this bit location. A bridge must implement this bit and the default state is 0 after reset.<br>0 - Target-Abort not signaled by the bridge on its primary interface<br>1- Target-Abort signaled by the bridge on its primary interface |

[1] The exception is for configuration commands since these are not subtractive decoded. By specifying the slowest time of all devices on a bus segment, the subtractive decode agent may be able to move in the time in which subtractive decode can occur.

| | | |
|---|---|---|
| 12 | Received Target-Abort | This bit reports the detection of a Target-Abort termination by the bridge when it is the master of a transaction on its primary interface. Once set, this bit remains set until it is reset by writing a 1 to this bit location. A bridge must implement this bit and the default state is 0 after reset.<br>0 - Target-Abort not detected by the bridge on its primary interface<br>1 - Target-Abort detected by the bridge on its primary interface |
| 13 | Received Master-Abort | This bit reports the detection of a Master-Abort termination by the bridge, when it is the master of a transaction on its primary interface. Once set, this bit remains set until it is reset by writing a 1 to this bit location. A bridge must implement this bit and the default state of this bit is 0 after reset.<br>0 - Master-Abort not detected by the bridge on its primary interface<br>1 - Master-Abort detected by the bridge on its primary interface |
| 14 | Signaled System Error | This bit reports the assertion of **SERR#** by the bridge on its primary interface. Once set, this bit remains set until it is reset by writing a 1 to this bit location. A bridge must implement this bit and the default state is 0 after reset.<br>0 - **SERR#** not asserted by the bridge on its primary interface<br>1 - **SERR#** asserted by the bridge on its primary interface |

| | | |
|---|---|---|
| 15 | Detected Parity Error | This bit reports the detection of an address or data parity error by the bridge on its primary interface. This bit must be set when any of the following three conditions is true:<br>• Detects an address parity error as a potential target;<br>• Detects a data parity error when the target of a write transaction; or<br>• Detects a data parity error when the master of a read transaction.<br>The bit is set regardless of the state of the Parity Error Response bit (bit 6) in the Command register. Once set, this bit remains set until it is reset by writing a 1 to this bit location. A bridge must implement this bit and the default state is 0 after reset.<br>0 - address or data parity error not detected by the bridge on its primary interface<br>1- address or data parity error detected by the bridge on its primary interface |

## 3.2.4.5. Revision ID Register

The Revision ID register specifies a device-specific revision identifier. The Revision ID is allocated by the vendor of the bridge device and must be implemented as a read-only register.

## 3.2.4.6. Class Code Register

The Class Code register is used to identify the function of the device and is broken into three byte-wide fields: base class code, sub-class code, and programming interface. A bridge returns a value of 060400h or 060401h when this register is read indicating a base class of 06h (bridge device), a sub-class code of 04h (PCI-to-PCI bridge), and a programming interface of 00h or 01h. The programming interface code of 00h is assigned to a bridge that supports the requirements of this specification.

The programming interface code of 01h is assigned to a bridge that supports the requirements of this specification plus it also supports subtractive decoding on the primary interface (see the Device Selection section of the *PCI Local Bus Specification* for more information on subtractive decoding). A device-specific configuration bit is permitted to enable and disable subtractive decoding. If such a bit is implemented and subtractive decoding is enabled, the programming interface is also set to 01h. When subtractive decoding is disabled, the programming interface is set to 00h.

**Implementation Note: Subtractive Decoding PCI-to-PCI Bridge**

The primary use of a subtractive decoding bridge is to connect a laptop system to a docking station and support legacy ISA devices in the docking station. The following is a list of some uses and restrictions of a subtractive decoding bridge:

- There can be only one subtractive decoding device on a PCI bus segment.
- A subtractive decoding bridge can support legacy programmed IO devices (like ISA) on the secondary bus.
- An ISA DMA controller requires DMA request and grant signals between it and the ISA DMA slave devices. The ISA DMA controller does not support a Retry capability and therefore no posted writes can exist between the DMA Slave and system memory when a DMA read is executed. A description of the requirements for supporting an ISA DMA slave downstream of a subtractive decoding PCI-to-PCI bridge is beyond the scope of this specification.
- The only unique feature indicated by class code 060401h is the support of subtractive decoding in addition to all the other requirements of the *PCI-to-PCI Bridge Architecture Specification*. No other functional characteristics should be assumed of bridges that have a 060401h class code.
- The purpose of the subtractive decoding class code is to let configuration software that configures the PCI devices know the bridge supports subtractive decoding.

## 3.2.4.7. Cacheline Size Register

The Cacheline Size register is used when terminating a transaction that uses the Memory Write and Invalidate command and when prefetching (Memory Read Line and Memory Read Multiple commands). Note that only cacheline sizes that are a power of two are valid. A bridge is permitted to limit the number of cacheline sizes that it supports. For example, it may accept cacheline sizes that are a power of 2 that are less than 128 bytes. If an unsupported value is written to the Cacheline Size register, the bridge must behave as if a value of 0 was written.

The Cacheline Size register must be implemented as a read/write register in a bridge that originates or forwards the Memory Write and Invalidate transaction (see Section 5.2.1. for details). If the bridge does not originate or forward Memory Write and Invalidate transactions, or support any of the enhanced memory read commands, the Cacheline Size register must be implemented as a read-only register that returns 0 when read.

A detailed description of the operation of the Cacheline Size register is provided in the *PCI Local Bus Specification*.

### 3.2.4.8. Latency Timer Register

The Latency Timer register is required if a bridge is capable of a burst transfer of more than two data phases on its primary interface. If implemented, the Latency Timer register must be a read/write register, and the implementation is permitted to limit the granularity to eight PCI clocks by hardwiring the low three bits to 0. A bridge that is not capable of a burst transfer of more than two data phases on its primary interface is permitted to hardwire the Latency Timer to a value of 16 or less. If implemented as a read/write register, the default value of the Latency Timer after reset must be 0. A detailed description of the operation of the Latency Timer is provided in the *PCI Local Bus Specification*.

### 3.2.4.9. Header Type Register

The Header Type register is a read-only register used to indicate the layout for bytes 10h through 3Fh of the device's configuration space. A bridge returns a value of 01h to indicate that the header adheres to the PCI-to-PCI bridge Configuration Space layout defined by this specification. If a bridge is a multi-function device (i.e., it integrates other functions besides the bridge), then a value of 81h is returned when the Header Type register is read.

### 3.2.4.10. BIST Register

The BIST register is an optional register used for control and status reporting of built-in self test capability. A bridge that does not support BIST must implement this register as a read-only register that returns 0 when read. Table 3-4 defines the bits in the BIST register when the bridge supports BIST. A bridge whose BIST is invoked must not interfere with normal operation of other devices on the primary bus, but the bridge will not respond as a target (except for configuration accesses to the BIST register) nor forward any transactions to the opposite bus during BIST. The effect BIST has on the secondary bus is not specified. Software cannot rely on any behavior of secondary bus devices or bridge functions (other than configuration register access of the BIST register) while BIST is active. After BIST completes the bridge and all downstream devices must be reinitialized by software.

**Table 3-4: BIST Register**

| BIST Register Bit | Bit Function | PCI-to-PCI Bridge Specific Notes |
|---|---|---|
| 3::0 | BIST Result | This bit field reports the result of the BIST operation. A value of zero means the device passed its test. Non-zero values mean the device failed. Device-specific failure codes assigned by the vendor are permitted to be encoded in the non-zero value. |
| 5::4 | reserved | These bits are reserved for future use by the PCI SIG and must be implemented as read-only and return 0 when read. |

| | | |
|---|---|---|
| 6 | Start BIST | This bit is used to initiate the BIST operation in the device and to indicate the completion of the BIST operation. Writing a 1 to this bit location initiates the BIST operation. The bridge resets this bit to 0 to indicate that it has completed the BIST operation. Software is permitted to assume the device has failed, if BIST is not completed after 2 seconds.<br>0 - BIST operation complete<br>1 - BIST operation in progress |
| 7 | BIST Capable | This bit is read-only and returns 1 when read, if the bridge supports BIST. If the bridge does not support BIST, then this bit and bits<6:0> must return 0 when read.<br>0 - BIST not supported<br>1 - BIST supported |

## 3.2.5. Bridge Specific Configuration Registers

## 3.2.5.1. Base Address Registers

The Base Address registers are optional registers used to map internal (device-specific) registers into Memory or I/O Spaces. These registers have no effect on the forwarding of transactions across a bridge as specified by the I/O, Memory, and Prefetchable Memory Base and Limit registers. Configuration software must map address ranges requested by the Base Address registers such that they are exclusive of the address ranges specified by the I/O, Memory, and Prefetchable Memory Base and Limit registers. Device-specific registers mapped by the Base Address registers must be accessible from both interfaces of the bridge.

Note that the bridge configuration header layout only provides two Base Address registers. As a consequence, if a 64-bit memory mapping is needed, then only one Base Address range can be supported (both Base Address registers will be consumed by the single 64-bit mapping). It is recommended that the bridge map its internal resources into a 32-bit address space.

If the bridge implements a single 32-bit Base Address register, the bridge is permitted to use either location. Base Address registers not used must be implemented as read-only registers that return 0 when read. Base Address registers must adhere to either the memory base address register or the I/O base address register format. The following sections give a brief overview of the bit definitions and typical usage. Refer to the *PCI Local Bus Specification* for a detailed discussion of Base Address registers.

## 3.2.5.1.1. Memory Base Address Register Format

If a bridge implements a Base Address register to map internal device-specific registers to a memory address range, it must adhere to the following register format.

**Table 3-5: Memory BAR Register**

| Memory Base Address Register Bit | Bit Function | PCI-to-PCI Bridge Specific Notes |
|---|---|---|
| 0 | Memory Space Indicator | This bit is implemented as a read-only bit that returns 0 when read.<br>This value indicates a range of memory addresses is being requested (if a Base Address register is supported). |
| 2::1 | Memory Mapping Type | This read-only bit field encodes the attributes of the memory address range requested by the Base Address register as listed below.<br>00 - Base Address register is 32 bits wide and may be mapped anywhere in the 32-bit memory space.<br>01 - this encoding is not allowed in a bridge<br>*Note: the 01 encoding was supported in prior revisions of the PCI Local Bus Specification to explicitly request mapping of the memory resource below the first 1 MB boundary. The intent of bridge architecture defined by this document is that the bridge be capable of operation downstream of another bridge. As a consequence, memory resources must be capable of being mapped above the first 1 MB boundary.*<br>10 - Base Address register is 64 bits wide and can be mapped anywhere in the 64-bit address space<br>11 - reserved |

| | | |
|---|---|---|
| 3 | Prefetchable | This read-only bit indicates the prefetchability of the requested memory address range. If set, the memory address range is prefetchable (i.e., has no read side effects and returns all bytes on reads regardless of the byte enables) and byte merging of write transactions is allowed. If cleared, the memory address range is not prefetchable and may have read side effects.<br>0 - not prefetchable<br>1 - prefetchable |
| 31::4 | Base Address | This bit field is used to indicate the size of the requested address range and to map the range to a specific set of addresses. The usage of the Base Address bit field is explained in detail in the *PCI Local Bus Specification*. The default value for the Base Address bit field after reset is undefined, since it must be written by software during the configuration process to determine the resource being requested. |

## 3.2.5.1.2. I/O Base Address Register Format

If a bridge implements a Base Address register to map internal device-specific registers to an I/O address range, it must adhere to the following register format.

**Table 3-6: I/O BAR Register**

| Memory Base Address Register Bit | Bit Function | PCI-to-PCI Bridge Specific Notes |
|---|---|---|
| 0 | I/O Space Indicator | This bit is implemented as a read-only bit that returns a 1 when read.<br>This value indicates a range of I/O addresses is being requested. |
| 1 | reserved | This bit is reserved for future use and must be implemented as a read-only bit that returns 0 when read. |
| 31::2 | Base Address | This bit field is used to indicate the size of the requested address range and to map the range to a specific set of addresses. The usage of the Base Address bit field is explained in detail in the *PCI Local Bus Specification*. The default value for the Base Address bit field after reset is undefined since it must be written by software during the configuration process. |

### 3.2.5.2. Primary Bus Number Register

The Primary Bus Number register is used to record the bus number of the PCI bus segment to which the primary interface of the bridge is connected. Configuration software programs the value in this register. The bridge uses this register to decode Type 1 configuration transactions on the secondary interface that must be converted to Special Cycle transactions on the primary interface. A bridge must[2] implement this register as a read/write register and the default state after reset must be zero.

### 3.2.5.3. Secondary Bus Number Register

The Secondary Bus Number register is used to record the bus number of the PCI bus segment to which the secondary interface of the bridge is connected. Configuration software programs the value in this register. The bridge uses this register to determine when to respond to a Type 1 configuration transaction on the primary interface and convert it to a Type 0 transaction on the secondary interface. The bridge also uses the Secondary Bus Number register and the Subordinate Bus Number register to determine when to forward Type 1 configuration transactions upstream. A bridge must implement this register as a read/write register and the default state after reset must be zero.

### 3.2.5.4. Subordinate Bus Number Register

The Subordinate Bus Number register is used to record the bus number of the highest numbered PCI bus segment which is behind (or subordinate to) the bridge. Configuration software programs the value in this register. The bridge uses this register in conjunction with the Secondary Bus Number register to determine when to respond to a Type 1 configuration transaction on the primary interface and to pass it to the secondary interface. The bridge also uses the Secondary Bus Number register and the Subordinate Bus Number register to determine when to forward Type 1 configuration transactions upstream. A bridge must implement this register as a read/write register and the default state after reset must be zero.

### 3.2.5.5. Secondary Latency Timer Register

The Secondary Latency Timer register adheres to the definition of the Latency Timer in the *PCI Local Bus Specification* but applies only to the secondary interface of a bridge. A bridge that supports a burst transfer of more than two data phases on its secondary interface must implement the Secondary Latency Timer as a read/write register. The implementation is permitted to limit the granularity to eight PCI clocks by hardwiring the low three bits to 0. A bridge which does not support a burst transfer of more than two data phases on its secondary interface is permitted

[2] An exception is granted to a bridge that is always connected to bus segment 0 by design or implementation. For example, this occurs when a PCI to PCI bridge is integrated into a host bus bridge that generates bus segment 0. In this case the bridge may implement this register as a read-only register that returns 0 when read.

to hardwire the Secondary Latency Timer to a value of 16 or less. If implemented as a read/write register, the default value of the Secondary Latency Timer after reset must be 0.

## 3.2.5.6. I/O Base Register and I/O Limit Register

The I/O Base and I/O Limit registers are optional and define an address range that is used by the bridge to determine when to forward I/O transactions from one interface to the other.

If a bridge does not implement an I/O address range, then both the I/O Base and I/O Limit registers must be implemented as read-only registers that return zero when read. If a bridge supports an I/O address range, then these registers must be initialized by configuration software so default states are not specified.

If a bridge implements an I/O address range, the upper 4 bits of both the I/O Base and I/O Limit registers are writable and correspond to address bits **AD[15::12]**. For the purpose of address decoding, the bridge assumes that the lower 12 address bits, **AD[11::00]**, of the I/O base address (not implemented in the I/O Base register) are zero. Similarly, the bridge assumes that the lower 12 address bits, **AD[11::00]**, of the I/O limit address (not implemented in the I/O Limit register) are FFFh. Thus, the bottom of the defined I/O address range will be aligned to a 4 KB boundary and the top of the defined I/O address range will be one less than a 4 KB boundary.

The I/O Limit register can be programmed to a smaller value than the I/O Base register, if there are no I/O addresses on the secondary side of the bridge. In this case, the bridge will not forward any I/O transactions from the primary bus to the secondary and will forward all I/O transactions from the secondary bus to the primary bus.

The lower four bits of both the I/O Base and I/O Limit registers are read-only, contain the same value, and encode the I/O addressing capability of the bridge according to Table 3-7.

**Table 3-7: I/O Adressing Capability**

| Register Bits [3::0] | I/O Addressing Capability |
|---|---|
| 00h | 16 bit I/O addressing |
| 01h | 32 bit I/O addressing |
| 02h - 0Fh | reserved |

If the low four bits of the I/O Base and I/O Limit registers have the value 0h, then the bridge supports only 16-bit I/O addressing (for ISA compatibility), and for the purpose of address decoding, the bridge assumes that the upper 16 address bits, **AD[31::16]**, of the I/O base and I/O limit address (not implemented in the I/O Base and I/O Limit registers) are zero. Note that the bridge must still perform a full 32-bit decode of the I/O address as required by the *PCI Local Bus Specification* (i.e., check that **AD[31::16]** are 0000h). In this case, the I/O address range supported by the bridge will be restricted to the first 64 KB of I/O Space (0000 0000h to 0000 FFFFh).

If the low four bits of the I/O Base and I/O Limit registers are 01h, then the bridge supports 32-bit I/O address decoding, and the I/O Base Upper 16 Bits and the I/O Limit Upper 16 Bits hold the upper 16 bits, corresponding to **AD[31::16]**, of the 32-bit I/O Base and I/O Limit

addresses respectively. In this case, system configuration software is permitted to locate the I/O address range supported by the anywhere in the 4-GB I/O Space. Note that the 4-KB alignment and granularity restrictions still apply when the bridge supports 32-bit I/O addressing.

## 3.2.5.7. Secondary Status Register

The Secondary Status register is similar in function and bit definition to the Status register defined in the *PCI Local Bus Specification*; however, its bits reflect status conditions of the secondary interface (the Status register reflects the status conditions of the primary interface).

The notable difference between the Status register bit definitions and the Secondary Status register bit definitions is that the Signaled System Error bit (which is bit 14 and is described in Table 3-8) has been redefined to be the Received System Error bit (for the secondary interface). See Section 6.6. for additional information on the assertion of **SERR#** on the secondary interface of a bridge.

**Table 3-8: Secondary Status Registers**

| Secondary Status Register Bit | Bit Function | Description |
|---|---|---|
| 4::0 | reserved | These bits are reserved for future use by the PCI SIG and must be implemented as read-only bits, which return 0 when read. |
| 5 | 66 MHz Capable | This bit indicates whether or not the secondary interface of the bridge is capable of operating at 66 MHz. Support of 66 MHz operation by a bridge is optional. This bit must be implemented as a read-only bit.<br>0 - the secondary interface of the bridge is not capable of 66 MHz operation<br>1 - the secondary interface of the bridge is capable of 66 MHz operation |
| 6 | reserved | This bit is reserved for future use by the PCI SIG and must be implemented as a read-only bit, which returns 0 when read. |

| | | |
|---|---|---|
| 7 | Fast Back-to-Back Capable | This bit indicates whether or not the secondary interface of the bridge is capable of decoding fast back-to-back transactions when the transactions are from the same master but to different targets. (A bridge is required to support fast back-to-back transactions from the same master.) This bit must be implemented as a read-only bit.<br>0 - the secondary interface of the bridge is not capable of decoding fast back-to-back transactions to different targets<br>1 - the secondary interface of the bridge is capable of decoding fast back-to-back transaction to different targets |
| 8 | Master Data Parity Error | This bit is used to report the detection of a parity error by the bridge when it is the master of a transaction. This bit is set if the following three conditions are all true:<br>• The bridge is the bus master of the transaction on the secondary interface;<br>• The bridge asserted **PERR#** (read transaction) or detected **PERR#** asserted (write transaction); and<br>• The Parity Error Response bit in the Bridge Control register is set.<br>Once set, this bit remains set until it is reset by writing a 1 to this bit location. A bridge must implement this bit and the default state must be 0 after reset.<br>0 - no parity error detected on the secondary interface<br>1 - parity error detected on the secondary interface |
| 10::9 | **DEVSEL#** Timing | This read-only bit field encodes the timing of the secondary interface **DEVSEL#** as listed below. The encoding must indicate the slowest response time that the bridge uses to assert **DEVSEL#** on its secondary interface when it is responding as a target to any transaction except a Configuration Read or Configuration Write.<br>00 - fast **DEVSEL#** decoding<br>01 - medium **DEVSEL#** decoding<br>10 - slow **DEVSEL#** decoding<br>11 - reserved |

| | | |
|---|---|---|
| 11 | Signaled Target-Abort | This bit reports the signaling of a Target-Abort termination by the bridge when it responds as the target of a transaction on its secondary interface. Once set, this bit remains set until it is reset by writing a 1 to this bit location. A bridge must implement this bit and the default state must be 0 after reset.<br>0 - Target-Abort not signaled by the bridge on its secondary interface<br>1- Target-Abort signaled by the bridge on its secondary interface |
| 12 | Received Target-Abort | This bit reports the detection of a Target-Abort termination by the bridge when it is the master of a transaction on its secondary interface. Once set, this bit remains set until it is reset by writing a 1 to this bit location. A bridge must implement this bit. The default state of this bit must be 0 after reset.<br>0 - Target-Abort not detected by the bridge on its secondary interface<br>1 - Target-Abort detected by the bridge on its secondary interface |
| 13 | Received Master-Abort | This bit reports the detection of a Master-Abort termination by the bridge when it is the master of a transaction on its secondary interface. Once set, this bit remains set until it is reset by writing a 1 to this bit location. A bridge must implement this bit and the default state must be 0 after reset.<br>0 - Master-Abort not detected by the bridge on its secondary interface<br>1 - Master-Abort detected by the bridge on its secondary interface |
| 14 | Received System Error | This bit reports the detection of an **SERR#** assertion on the secondary interface of the bridge. Once set, this bit remains set until it is reset by writing a 1 to this bit location. The default state must be 0 after reset.<br>0 - **SERR#** assertion on the secondary interface has not been detected<br>1 - **SERR#** assertion on the secondary interface has been detected |

| | | |
|---|---|---|
| 15 | Detected Parity Error | This bit reports the detection of an address or data parity error by the bridge on its secondary interface. This bit must be set when any of the following three conditions is true:<br>• Detects an address parity error as a potential target;<br>• Detects a data parity error when the target of a write transaction; or<br>• Detects a data parity error when the master of a read transaction.<br>The bit is set regardless of the state of the Parity Error Response Enable bit (bit 0) in the Bridge Control register. Once set, this bit remains set until it is reset by writing a 1 to this bit location. A bridge must implement this bit, and the default state must be 0 after reset.<br>0 - address or data parity error not detected by the bridge on its secondary interface<br>1- address or data parity error detected by the bridge on its secondary interface |

## 3.2.5.8. Memory Base Register and Memory Limit Register

The Memory Base and Memory Limit registers are both required registers that define a memory mapped I/O address range which is used by the bridge to determine when to forward memory transactions from one interface to the other (see Section 4.3. for additional details).

The upper 12 bits of both the Memory Base and Memory Limit registers are read/write and correspond to the upper 12 address bits, **AD[31::20]**, of 32-bit addresses. For the purpose of address decoding, the bridge assumes that the lower 20 address bits, **AD[19::00]**, of the memory base address (not implemented in the Memory Base register) are zero. Similarly, the bridge assumes that the lower 20 address bits, **AD[19::00]**, of the memory limit address (not implemented in the Memory Limit register) are F FFFFh. Thus, the bottom of the defined memory address range will be aligned to a 1 MB boundary and the top of the defined memory address range will be one less than a 1 MB boundary.

The Memory Limit register must be programmed to a smaller value than the Memory Base register if there are no memory-mapped I/O addresses on the secondary side of the bridge. If there is no prefetchable memory (see Section 3.2.5.9.), and there is no memory-mapped I/O on the secondary side of the bridge, then the bridge will not forward any memory transactions from the primary bus to the secondary bus and will forward all memory transactions from the secondary bus to the primary bus.

The bottom four bits of both the Memory Base and Memory Limit registers are read-only and return zeros when read.

These registers must be initialized by configuration software so default states are not specified.

## 3.2.5.9. Prefetchable Memory Base Register and Prefetchable Memory Limit Register

The Prefetchable Memory Base and Prefetchable Memory Limit registers are optional. They define a prefetchable memory address range which is used by the bridge to determine when to forward memory transactions from one interface to the other (see Section 4.4. for additional details).

If a bridge does not implement a prefetchable memory address range, then both Prefetchable Memory Base and Prefetchable Memory Limit registers must be implemented as read-only registers which return zero when read. If a bridge implements a prefetchable memory address range, then both of these registers must be implemented as read/write registers. If a bridge supports a prefetchable memory address range, then these registers must be initialized by configuration software so default states are not specified.

If the bridge implements a prefetchable memory address range, the upper 12 bits of the register are read/write and correspond to the upper 12 address bits, **AD[31::20]**, of 32-bit addresses. For the purpose of address decoding, the bridge assumes that the lower 20 address bits, **AD[19::00]**, of the prefetchable memory base address (not implemented in the Prefetchable Memory Base register) are zero. Similarly, the bridge assumes that the lower 20 address bits, **AD[19::00]**, of the prefetchable memory limit address (not implemented in the Prefetchable Memory Limit register) are F FFFFh. Thus, the bottom of the defined prefetchable memory address range will be aligned to a 1 MB boundary and the top of the defined memory address range will be one less than a 1 MB boundary.

The Prefetchable Memory Limit register must be programmed to a smaller value than the Prefetchable Memory Base register if there is no prefetchable memory on the secondary side of the bridge. If there is no prefetchable memory, and there is no memory-mapped I/O (see Section 3.2.5.8.) on the secondary side of the bridge, then the bridge will not forward any memory transactions from the primary bus to the secondary and will forward all memory transactions from the secondary bus to the primary bus.

The bottom 4 bits of both the Prefetchable Memory Base and Prefetchable Memory Limit registers are read-only, contain the same value, and encode whether or not the bridge supports 64-bit addresses. If these four bits have the value 0h, then the bridge supports only 32 bit addresses. If these four bits have the value 01h, then the bridge supports 64-bit addresses and the Prefetchable Base Upper 32 Bits and Prefetchable Limit Upper 32 Bits registers hold the rest of the 64-bit prefetchable base and limit addresses respectively. All other encodings are reserved.

## 3.2.5.10. Prefetchable Base Upper 32 Bits and Prefetchable Limit Upper 32 Bits Registers

The Prefetchable Base Upper 32 Bits and Prefetchable Limit Upper 32 Bits registers are optional extensions to the Prefetchable Memory Base and Prefetchable Memory Limit registers.

If the Prefetchable Memory Base and Prefetchable Memory Limit registers indicate support for 32-bit addressing, then the Prefetchable Base Upper 32 Bits and Prefetchable Limit Upper 32 Bits registers are both implemented as read-only registers that return zero when read. If the Prefetchable Memory Base and Prefetchable Memory Limit registers indicate support for 64-bit

addressing, then the Prefetchable Base Upper 32 Bits and Prefetchable Limit Upper 32 Bits registers are implemented as read/write registers. If these registers are implemented as read/write registers, they must be initialized by configuration software so default states are not specified.

If a 64-bit prefetchable memory address range is supported, the Prefetchable Base Upper 32 Bits and Prefetchable Limit Upper 32 Bits registers specify the upper 32-bits, corresponding to **AD[63::32]**, of the 64-bit base and limit addresses which specify the prefetchable memory address range (see Section 4.4.2. for additional details).

## 3.2.5.11. I/O Base Upper 16 Bits and I/O Limit Upper 16 Bits Registers

The I/O Base Upper 16 Bits and I/O Limit Upper 16 Bits registers are optional extensions to the I/O Base and I/O Limit registers. If the I/O Base and I/O Limit registers indicate support for 16-bit I/O address decoding, then the I/O Base Upper 16 Bits and I/O Limit Upper 16 Bits registers are implemented as read-only registers which return zero when read. If the I/O Base and I/O Limit registers indicate support for 32-bit I/O addressing, then the I/O Base Upper 16 Bits and I/O Limit Upper 16 Bits registers must be initialized by configuration software so default states are not specified.

If 32-bit I/O address decoding is supported, the I/O Base Upper 16 Bits and the I/O Limit Upper 16 Bits register specify the upper 16 bits, corresponding to **AD[31::16]**, of the 32-bit base and limit addresses respectively, that specify the I/O address range (see Section 4.2. for additional details).

## 3.2.5.12. Capabilities Pointer

This optional register is used to point to a linked list of additional capabilities implemented by this device. The Capabilities List Section and Capability IDs Appendix in the *PCI Local Bus Specification* specify the linked list data structure and a list of defined capabilities, respectively.

If the Capabilities List bit (bit 4) in the Status register is zero, then the default state of this register is zero after reset. However, if subsequently written, the register may return an indeterminate value. Therefore, software is not permitted to write this register.

If the Capabilities List bit (bit 4) in the Status register is set, then this register must be implemented as a read-only register. The bottom two bits of the pointer are reserved for future use and must be set to 00b. However, software cannot depend on them being zero and must mask them off before using this register as a pointer to a configuration register which holds the first entry of a linked list of new capabilities.

## 3.2.5.13. Reserved Registers at 35h, 36h, and 37h

Read accesses to these registers must complete normally and return a value of zero after reset. However, if subsequently written, these registers may return an indeterminate value.

## 3.2.5.14. Expansion ROM Base Address Register

The Expansion ROM Base Address register is an optional register that adheres to the definition contained in the *PCI Local Bus Specification*. Note, however, that the offset of the register within the Type 1 configuration header for a bridge is different than that of the Type 0 configuration header specified in the *PCI Local Bus Specification*.

## 3.2.5.15. Interrupt Line Register

The Interrupt Line register is a read/write register used to communicate interrupt line routing information between initialization code and the device driver. This register must be initialized by initialization code so a default state is not specified. The value written to the Interrupt Line register specifies the routing of the device's **INTx#** pin to the system interrupt controller. If a bridge does not implement an interrupt signal pin, then POST (power-on self test) code must write FFh to this register.

## 3.2.5.16. Interrupt Pin Register

The Interrupt Pin register is a read-only register that adheres to the definition in the *PCI Local Bus Specification*. The Interrupt Pin register is used to indicate which interrupt pin the bridge uses. A value of 1 corresponds to **INTA#**. A value of 2 corresponds to **INTB#**. A value of 3 corresponds to **INTC#**. A value of 4 corresponds to **INTD#.**

If a bridge is not a multi-function device, it may only use **INTA#** as its interrupt pin (if implemented). A bridge that does not implement any interrupt pins must return a 0 for this register when read.

## 3.2.5.17. Bridge Control Register

The Bridge Control register provides extensions to the Command register that are specific to a bridge. The Bridge Control register provides many of the same controls for the secondary interface that are provided by the Command register for the primary interface. There are some bits that affect the operation of both interfaces of the bridge. Definitions for each bit are specified in Table 3-9.

**Table 3-9: Bridge Control Register**

| Bridge Control Register Bit | Bit Function | Description |
|---|---|---|
| 0 | Parity Error Response Enable | Controls the bridge's response to address and data parity errors on the secondary interface. If this bit is set, the bridge must take its normal action when a parity error is detected. If this bit is cleared, the bridge must ignore any parity errors that it detects and continue normal operation. A bridge must generate parity even if parity error reporting is disabled. The default state of this bit after reset must be 0.<br>0 - ignore address and data parity errors on the secondary interface<br>1 - enable parity error detection and reporting on the secondary interface |
| 1 | SERR# Enable | Controls the forwarding of secondary interface **SERR#** assertions to the primary interface. The bridge will assert **SERR#** on the primary interface when all of the following are true:<br>• **SERR#** is asserted on the secondary interface;<br>• This bit is set; and<br>• The SERR# Enable bit is set in the Command register.<br>The default state of this bit after reset must be 0.<br>0 - disable the forwarding of secondary **SERR#** to primary **SERR#**<br>1 - enable the forwarding of secondary **SERR#** to primary **SERR#** |

| | | |
|---|---|---|
| 2 | ISA Enable | Modifies the response by the bridge to ISA I/O addresses. This applies only to I/O addresses that are enabled by the I/O Base and I/O Limit registers and are in the first 64 KB of PCI I/O address space (0000 0000h to 0000 FFFFh). If this bit is set, the bridge will block any forwarding from primary to secondary of I/O transactions addressing the last 768 bytes in each 1 KB block. In the opposite direction (secondary to primary), I/O transactions will be forwarded if they address the last 768 bytes in each 1K block. The default state of this bit after reset must be 0.<br><br>0 - forward downstream all I/O addresses in the address range defined by the I/O Base and I/O Limit registers<br><br>1 - forward upstream ISA I/O addresses in the address range defined by the I/O Base and I/O Limit registers that are in the first 64 KB of PCI I/O address space (top 768 bytes of each 1 KB block) |

| | | |
|---|---|---|
| 3 | VGA Enable | Modifies the response by the bridge to VGA compatible addresses. If the VGA Enable bit is set, the bridge will positively decode and forward the following accesses on the primary interface to the secondary interface (and, conversely, block the forwarding of these addresses from the secondary to primary interface):<br>• memory accesses in the range 000A 0000h to 000B FFFFh<br>• I/O addresses in the first 64 KB of the I/O address space (**AD[31:16]** are 0000h) where **AD[9:: 0]** are in the ranges 3B0h to 3BBh and 3C0h to 3DFh (inclusive of ISA address aliases - **AD[15::10]** are not decoded)<br>If the VGA Enable bit is set, forwarding of these accesses is independent of the I/O address range and memory address ranges defined by the I/O Base and Limit registers, the Memory Base and Limit registers, and the Prefetchable Memory Base and Limit registers of the bridge. (Forwarding of these accesses is also independent of the settings of the ISA Enable bit (in the Bridge Control register) or VGA Palette Snoop bits (in the Command register), when the VGA Enable bit is set. Forwarding of these accesses is qualified by the I/O Enable and Memory Enable bits in the Command register.) The default state of this bit after reset must be 0.<br>0 - do not forward VGA compatible memory and I/O addresses from the primary to secondary interface (addresses defined above) unless they are enabled for forwarding by the defined I/O and memory address ranges<br>1 - forward VGA compatible memory and I/O addresses (addresses defined above) from the primary interface to the secondary interface (if the I/O Enable and Memory Enable bits are set) independent of the I/O and memory address ranges and independent of the ISA Enable bit |
| 4 | reserved | This bit is reserved for future use by the PCI SIG, is read-only, and must return zero when read. |

| | | |
|---|---|---|
| 5 | Master-Abort Mode | Controls the behavior of a bridge when a Master-Abort termination occurs on either interface while the bridge is the master of the transaction. The default state of this bit must be 0 after reset.<br><br>If the Master-Abort mode bit is cleared and a non-locked transaction that crosses the bridge terminates with a Master-Abort on the destination bus, reads will return all ones and write data will be accepted by the bridge and then discarded. See Section 6.3.3. for additional Master-Abort requirements during a locked transaction sequence.<br><br>If the Master-Abort Mode bit is set, the bridge signals a Target-Abort to the requesting master if the corresponding transaction on the other side of the bridge terminates with a Master-Abort and the transaction has not yet been concluded (reads and non-posted writes). If the transaction on the originating bus has completed (posted write), then the bridge must assert **SERR#** on the primary interface (provided the SERR# Enable bit is set in the Command register).<br><br>0 - do not report Master-Aborts (return FFFF FFFFh on reads and discard data on writes)<br><br>1 - report Master-Aborts by signaling Target-Abort if possible or by the assertion of **SERR#** (if enabled) |
| 6 | Secondary Bus Reset | Forces the assertion of **RST#** on the secondary interface. The secondary **RST#** will be asserted by the bridge whenever this bit is set or the **RST#** pin on the primary interface is asserted. When this bit is cleared, **RST#** on the secondary bus will be asserted whenever the primary interface **RST#** is asserted. The bridge's secondary bus interface and any buffers between the two interfaces (primary and secondary) must be initialized back to their default state whenever this bit is set. The primary bus interface and all configuration space registers must not be affected by the setting of this bit. The default state of this bit must be 0 after reset.<br><br>0 - do not force the assertion of the secondary interface **RST#**<br><br>1 - force the assertion of the secondary interface **RST#** |

| | | |
|---|---|---|
| 7 | Fast Back-to-Back Enable | Controls ability of the bridge to generate fast back-to-back transactions to different devices on the secondary interface. A bridge that cannot generate fast back-to-back transactions must implement this bit as a read-only bit that returns 0 when read. A bridge that is capable of initiating fast back-to-back transaction must implement this bit as a read/write bit with a default state of 0 after reset. During system initialization, configuration software will set this bit if all devices on the secondary interface are capable of fast back-to-back operation.<br>0 - disable generation of fast back-to-back transactions on the secondary interface<br>1 - enable generation of fast back-to-back transactions on the secondary interface |
| 8 | Primary Discard Timer | Selects the number of PCI clocks that the bridge will wait for a master on the primary interface to repeat a Delayed Transaction request (see Section 5.3.2. for more details). The counter starts once the Delayed Completion (the completion of the Delayed Transaction on the secondary interface) has reached the head of the upstream queue of the bridge (i.e., all ordering requirements have been satisfied and the bridge is ready to complete the Delayed Transaction with the originating master on the primary bus). If the originating master does not repeat the transaction before the counter expires, the bridge will delete the Delayed Transaction from its queue and set the Discard Timer Status bit. The default state of this bit after reset is 0.<br>0 - the Primary Discard Timer counts $2^{15}$ PCI clock cycles<br>1 - the Primary Discard Timer counts $2^{10}$ PCI clock cycles |

| | | |
|---|---|---|
| 9 | Secondary Discard Timer | Selects the number of PCI clocks that the bridge will wait for a master on the secondary interface to repeat a Delayed Transaction request (see Section 5.3.2. for more details). The counter starts once the Delayed Completion (the completion of the Delayed Transaction on the primary interface) has reached the head of the downstream queue of the bridge (i.e., all ordering requirements have been satisfied and the bridge is ready to complete the Delayed Transaction with the originating master on the secondary bus). If the originating master does not repeat the transaction before the counter expires, the bridge will delete the Delayed Transaction from its queue and set the Discard Timer Status bit. The default state of this bit after reset is 0.<br>0 - the Secondary Discard Timer counts $2^{15}$ PCI clock cycles<br>1 - The Secondary Discard Timer counts $2^{10}$ PCI clock cycles |
| 10 | Discard Timer Status | This bit is set to a 1 when either the Primary Discard Timer or Secondary Discard Timer expires and a Delayed Completion is discarded from a queue in the bridge. The default state of this bit after reset must be 0. Once set, this bit remains set until it is reset by writing a 1 to this bit location.<br>0 - no discard timer error<br>1 - discard timer error |
| 11 | Discard Timer SERR# Enable | When set to 1, this bit enables the bridge to assert **SERR#** on the primary interface when either the Primary Discard Timer or Secondary Discard Timer expires and a Delayed Transaction is discarded from a queue in the bridge. The default state of this bit must be 0 after reset.<br>0 - do not assert **SERR#** on the primary interface as a result of the expiration of either the Primary Discard Timer or Secondary Discard Timer<br>1 - assert **SERR#** on the primary interface if either the Primary Discard Timer or Secondary Discard Timer expires and a Delayed Transaction is discarded from a queue in the bridge |
| 15::12 | reserved | These bits are reserved for future use by the PCI SIG, are read-only, and must return zero when read. |

## 3.2.6. Slot Numbering Capabilities List Item

The slot numbering registers are optional. They are required for bridges that connect to PCI expansion chassis. Refer to Section 13.1. for additional details on slot numbering.

If the slot numbering registers are supported, the Capabilities List bit (bit 4 in the Status register) must be set to a 1, and the Slot Numbering Capabilities registers shown in Figure 3-3 must appear in the Capabilities List. The value stored in the Capabilities Pointer registers (offset 34h) points to the Slot Numbering Capabilities registers, if they are the first item in the Capabilities List. If not, a subsequent list item points to the Slot Numbering Capabilities registers.

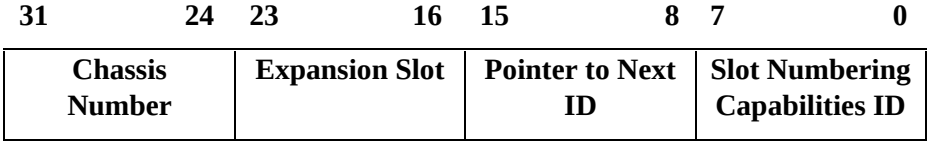



**Figure 3-3: Slot Numbering Capabilities Register**

### 3.2.6.1. Slot Numbering Capabilities ID

This register identifies the Capabilities List item as a Slot Numbering Registers item. It is read-only and returns the value of 04h when read.

### 3.2.6.2. Pointer to Next ID

This register contains the pointer to the next Capabilities List item, if supported. If there are no subsequent list items, this register will contain the value 0. This register is read-only.

### 3.2.6.3. Expansion Slot Register

The Expansion Slot register provides information used by system software in calculating the slot number of a device plugged into a PCI slot in an expansion chassis. Refer to Section 13.3. for a complete discussion of the use of the Expansion Slot register.

The register is read-only and is initialized by hardware after reset. The method by which the system designer establishes the default value of this register is not controlled by this specification, but the content must be valid when the PCI system initialization software reads the register to determine how the system is configured. Any alternative that guarantees the contents will be valid before the PCI system initialization software executes is acceptable. For example, the bridge could initialize the Expansion Slot register based on the state of certain pins on the bridge at **RST#** time. In this approach, the expansion chassis designer pulls these pins up or down based on how the expansion chassis is wired. After **RST#**, the pins assume their normal functions. In another example, the inputs to an external shift register could be hardwired with this information and the shift register read by the bridge at **RST#** time. More elaborate schemes involving serial EEPROMs would be possible as well.

**Table 3-10: Expansion Slot Register**

| Expansion Slot Register Bit | Bit Function | Description |
|---|---|---|
| 4::0 | Expansion Slots Provided | Contains the binary value of the number of PCI expansion slots located directly on the secondary interface of this bridge. Expansion slots located behind additional (subordinate) bridges on the secondary interface are not counted in this field. |
| 5 | First in Chassis | If this bit is set, it indicates that this bridge is the first in an expansion chassis. A bridge with this set indicates the existence of an expansion chassis that requires a unique chassis number.<br>0 - This is not a parent bridge.<br>1 - This is a parent bridge. |
| 7::6 | reserved | These bits are reserved for future use by the PCI SIG, are read only, and must return zero when read. |

## 3.2.6.4. Chassis Number Register

The Chassis Number register contains the physical chassis number for the slots on this bridge's secondary interface. A different non-zero chassis number is assigned by system initialization software to each separate expansion unit that contains PCI expansion slots. Multiple bridges contained in the same chassis are assigned the same chassis number. Chassis number 0 is reserved for the chassis containing the CPU that initializes the Configuration Space for the system.

This register is read/write and may optionally be either non-volatile or initialized to zero by reset. If the system configuration software finds the number in this register is non-zero and does not conflict with another chassis number, the system configuration software will leave the register value unchanged. If the system configuration software finds this register is zero or that it conflicts with another chassis number, the configuration software will write a new chassis number into this register.

The method of storage and retrieval of the non-volatile chassis number is not controlled by this specification, so any technology that holds information across a power cycle will suffice. For example, the register can be a simple read/write register with a battery backup. Alternatively, the chassis number can be stored in a serial EEPROM and shifted in at power up.

Making the Chassis Number register non-volatile provides the most capability to the end user of an expansion chassis. In this case, even if bridges to expansion chassis are rearranged in the system, the chassis number remains unchanged unless a chassis is moved from one system to another and causes a duplication of chassis numbers. Therefore, this alternative is recommended for bridges that have non-volatile storage capabilities.

Refer to Section 13.4. for a complete discussion of the use of the Chassis Number register.

# Chapter 4
# Address Decoding

## 4.1. Address Ranges

The configuration header for a bridge defines a set of base and limit registers for an optional I/O address range, a required memory mapped I/O address range, and an optional prefetchable memory address range. The base and limit address registers define the address ranges in which a bridge forwards transactions from its primary interface to its secondary interface. These registers are effectively inversely decoded to determine the address ranges on the secondary interface of a bridge in which transactions will be forwarded upstream (from the secondary to primary interface). A bridge does not perform any address translation (a flat addressing model is used). The following sections describe each of these address ranges in more detail. Optional support for VGA address decoding and subtractive decode are also described at the end of this chapter.

## 4.2. I/O

The optional I/O Base, I/O Limit, I/O Base Upper 16 Bits, and I/O Limit Upper 16 Bits registers in the bridge configuration header specify an address range that is used by the bridge to determine whether to forward I/O read and I/O write transactions across the bridge. Note that if the I/O Base is set to a value larger than the I/O Limit, the address range is disabled (see Section 3.2.5.6.). The response by the bridge to I/O transactions is also affected by the register bits listed below.

- I/O Enable bit in the Command register
- Master Enable bit in the Command register
- VGA Palette Snoop Enable bit in the Command register
- VGA Enable bit in the Bridge Control register
- ISA Enable bit in the Bridge Control register

The I/O Enable bit of the Command register must be set to enable the bridge's I/O address range (as required by the *PCI Local Bus Specification*). The VGA Enable Bit, VGA Palette Snoop

Enable bit, and the ISA Enable bit are discussed in later sections. More details can be found for all bits in the descriptions of the appropriate registers.

The I/O base and I/O limit registers are used by a bridge to determine whether to forward I/O transactions across the bridge as illustrated in Figure 4-1. The I/O address range defined by these registers is always aligned to a 4 KB boundary and has a size granularity of 4 KB. A bridge forwards I/O read and I/O write transactions from its primary interface to its secondary interface (downstream) if the address is in the range defined by the I/O base and I/O limit registers (when the base is less than or equal to the limit). Conversely, I/O transactions on the secondary bus in the address range defined by these registers are not forwarded upstream by the bridge. I/O transactions on the secondary bus that are outside the defined address range are forwarded upstream (from the secondary interface to the primary interface). If a bridge does not implement an I/O address range, the bridge must forward all I/O transactions on its secondary interface upstream to its primary interface.

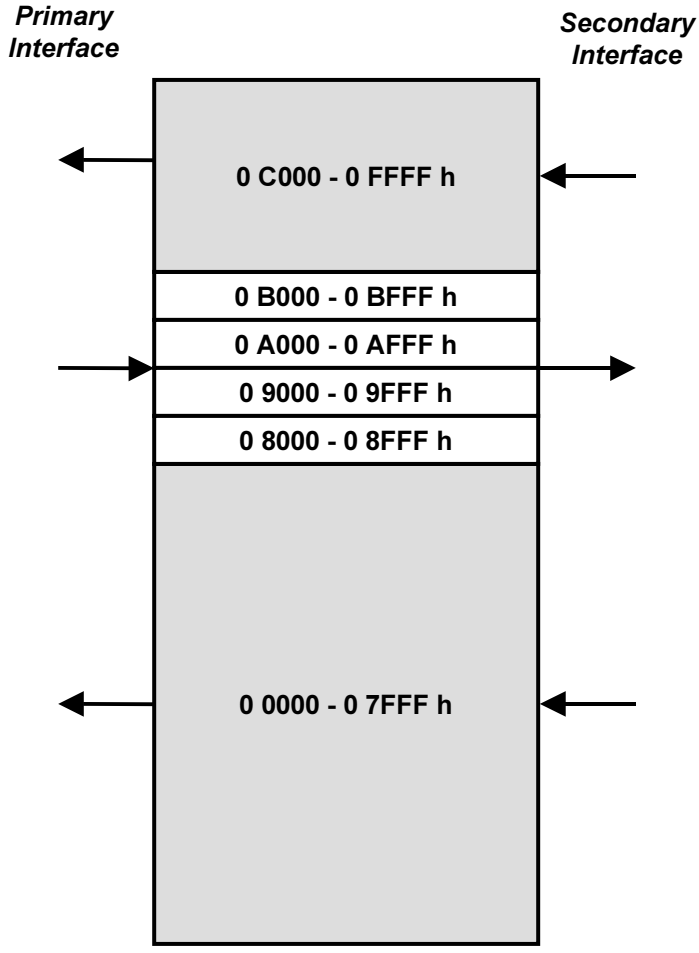



**Figure 4-1: I/O Address Range Example**

### 4.2.1. ISA Mode

Bridges also provide an ISA Enable bit in the Bridge Control register. This bit affects only I/O addresses that are in the bridge's I/O range (as defined by the I/O Base, I/O Base Upper 16 Bits, I/O Limit, and I/O Limit Upper 16 Bits) and in the first 64 KB of PCI I/O Space (0000 0000h to 0000 FFFFh). If this bit is set and the I/O address meets the stated constraints, the bridge will block forwarding of I/O transactions downstream (from the primary interface to the secondary interface) if the I/O address is in the top 768 bytes of each naturally aligned 1 KB block.

Conversely, if ISA addressing mode is enabled, I/O transactions on the secondary bus in the top 768 bytes of any 1 KB address block within the first 64 KB of PCI I/O Space will be forwarded upstream to the primary bus, even if the address is between the I/O base and I/O limit addresses. Figure 4-2 illustrates this mapping. The combination of the 4 KB granularity and the ISA Enable bit results in I/O address decoding on the secondary interface of the bridge that is similar to EISA slot decoding. Devices on the secondary interface are permitted to be mapped to the first 256 bytes of each naturally aligned 1 KB block within the defined I/O address range.

The ISA Enable bit only affects the I/O address decoding behavior of the bridge. It does not affect the bridge's prefetching, posting, ordering, or error handling behavior.

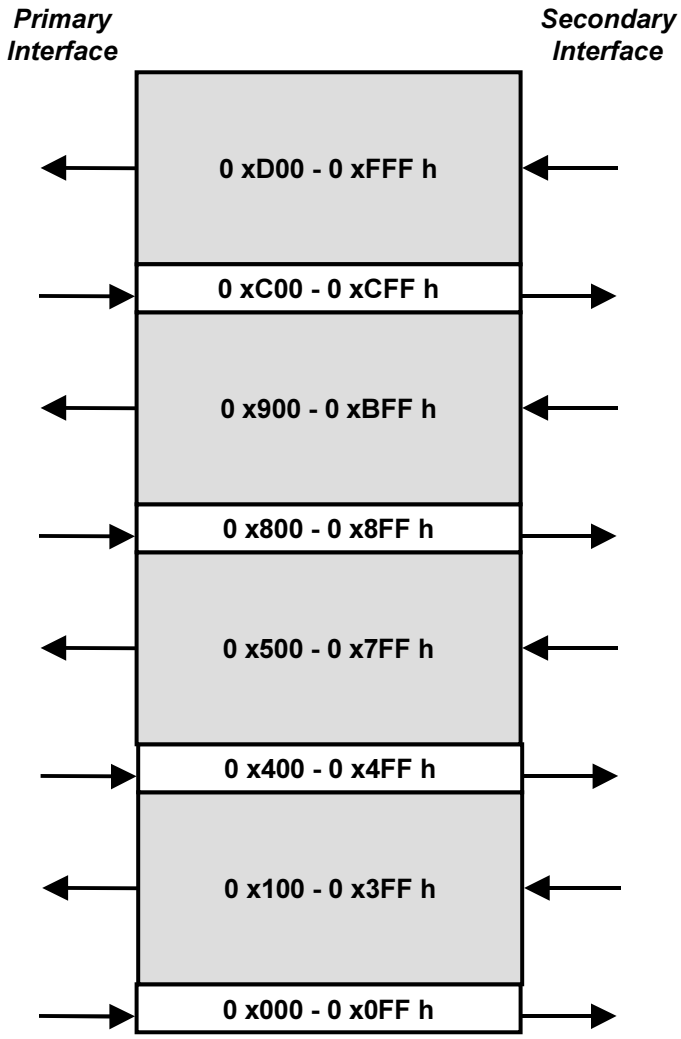



**Figure 4-2: ISA Mode I/O Address Range Example**

## 4.3. Memory Mapped I/O

The memory mapped I/O range is intended to be used to map memory address ranges of devices that are not prefetchable (i.e., have side effects on reads or non-memory-like behavior). The bridge will not prefetch read data if a transaction using the Memory Read command crosses the bridge in a memory mapped I/O address range (see Section 5.1.). Prefetching of read data is allowed in the memory mapped I/O address range when a transaction using the Memory Read Line or Memory Read Multiple commands is used. However, bridges are not required to support prefetching in the memory mapped I/O range and may choose to alias Memory Read Line and Memory Read Multiple commands to a Memory Read command.

The Memory Mapped I/O Base and Memory Mapped I/O Limit registers are required for a bridge. They are used by the bridge to determine whether to forward transactions using the Memory Read, Memory Read Line, Memory Read Multiple, Memory Write, and Memory Write and Invalidate commands across the bridge. Note that if the Memory Mapped I/O Base is set to

a value higher than the Memory Mapped I/O Limit, the address range is disabled (see Section 3.2.5.8.). The register bits listed below also affect the response by the bridge to memory transactions.

- Memory Enable bit in the Command register
- Master Enable bit in the Command register
- VGA Enable bit in the Bridge Control register

The Memory Enable bit in the bridge's Command register must be set to enable the memory mapped I/O address range. The VGA Enable Bit is discussed in Section 4.5.1. More details can be found for all bits in the descriptions of the appropriate registers. The prefetchable memory address range (see Section 3.2.5.9.) defined by the prefetchable memory address registers also affects response to memory transactions.

A bridge forwards PCI memory transactions from its primary interface to its secondary interface (downstream) if a memory address is in the range defined by the Memory Base and Memory Limit registers (when the base is less than or equal to the limit) as illustrated in Figure 4-3. Conversely, a memory transaction on the secondary interface that is within this address range will not be forwarded upstream to the primary interface. Any memory transactions on the secondary interface that are outside this address range will be forwarded upstream to the primary interface (provided they are not in the address range defined by the prefetchable memory address range registers).

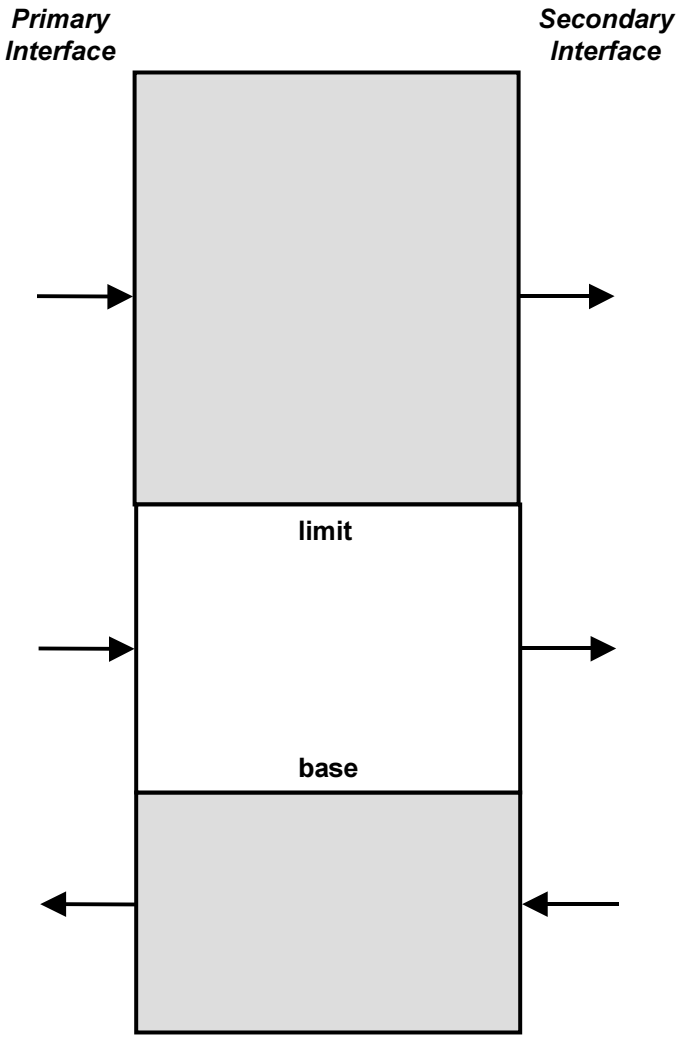



**Figure 4-3: Memory Address Range Example**

## 4.4. Prefetchable Memory

The optional prefetchable memory range is intended to be used to map memory address ranges of devices that are prefetchable (i.e., have memory-like behavior and do not have side effects on reads). The bridge can prefetch read data when a transaction using any memory read command (Memory Read, Memory Read Line, or Memory Read Multiple) crosses the bridge in a prefetchable memory address range (see Section 5.1.).

The optional Prefetchable Memory Base, Prefetchable Memory Limit, Prefetchable Base Upper 32-bits, and Prefetchable Limit Upper 32-bits registers in the bridge configuration header specify an address range that is used by the bridge to determine whether to forward transactions using Memory Read, Memory Read Line, Memory Read Multiple, Memory Write, and Memory Write and Invalidate commands across the bridge. Note that if the Prefetchable Memory Base register is set to a value higher than the Prefetchable Memory Limit register, the address range is disabled (see Section 3.2.5.9.). The register bits listed below also affect the response by the bridge to memory transactions.

- Memory Enable bit in the Command register
- Master Enable bit in the Command register
- VGA Enable bit in the Bridge Control register

The Memory Enable bit in the bridge's Command register must be set to enable the prefetchable memory address range. The VGA Enable Bit is discussed in Section 4.5.1. More details can be found for all bits in the descriptions of the appropriate registers. The memory mapped I/O address range (see Section 4.3.) defined by the Memory Base and Memory Limit registers also affects the response to memory transactions. A bridge forwards memory transactions from its primary interface to its secondary interface (downstream) if a memory address is in the range defined by the Prefetchable Memory Base and Prefetchable Memory Limit registers (when the base is less than or equal to the limit). Conversely, a memory transaction on the secondary interface that is within this address range will not be forwarded upstream to the primary interface. Any memory transactions on the secondary interface that are outside this address range will be forwarded upstream to the primary interface (provided they are not in the address range defined by the memory mapped I/O address range registers).

### 4.4.1. 64-bit Addressing

Bridges have numerous options regarding 64-bit addressing. The Dual Address Cycle (DAC) command defined by the *PCI Local Bus Specification* is used to implement 64-bit addressing on PCI. During a DAC, the high 32-bits of address, address bits **AD[63::32]**, are transmitted on **AD[31::00]** during the second address phase. This allows devices with only 32-bit data paths to utilize 64-bit addressing. DACs are used to access locations that are not in the first 4 GB of PCI Memory Space. Addresses in the first 4 GB region of Memory Space always use a single address cycle (SAC) and never use a DAC.

### Implementation Note: 64-bit Addressing

In the simplest solution, a bridge may chose to ignore Dual Address Cycles on its primary PCI bus and not forward any DAC commands downstream. In this case, the 64-bit extensions of the Prefetchable Memory Base and Prefetchable Memory Limit registers (upper 32-bits) are not implemented. This solution assumes that no targets located downstream of the bridge support 64-bit addressing.

If Dual Address Cycles are not supported on the primary interface of the bridge, the bridge must forward all DAC commands upstream (from its secondary to primary interface). This allows masters downstream of a bridge to access system memory that may be located above the first 4 GB. The only targets that seem likely to require 64-bit addressing would be devices that implement large buffers that would typically have memory-like behavior and indicate in their memory base register that they are prefetchable. Bridges can optionally support these devices when they are downstream of the bridge by implementing the prefetchable memory space registers including the Prefetchable Base Upper 32-bits and Prefetchable Limit Upper 32-bits registers. In this case, inverse decoding is used by the bridge to determine when to forward Dual Address Cycles upstream.

Figure 4-4 shows how Dual Address Cycle transactions will be forwarded by the bridge from the primary interface to the secondary interface when they are in the 64-bit prefetchable memory address range (when the base is less than or equal to the limit). Dual Address Cycle transactions will be forwarded by the bridge from the secondary interface to the primary interface when the address is outside the range defined by the prefetchable memory base and limit address registers.

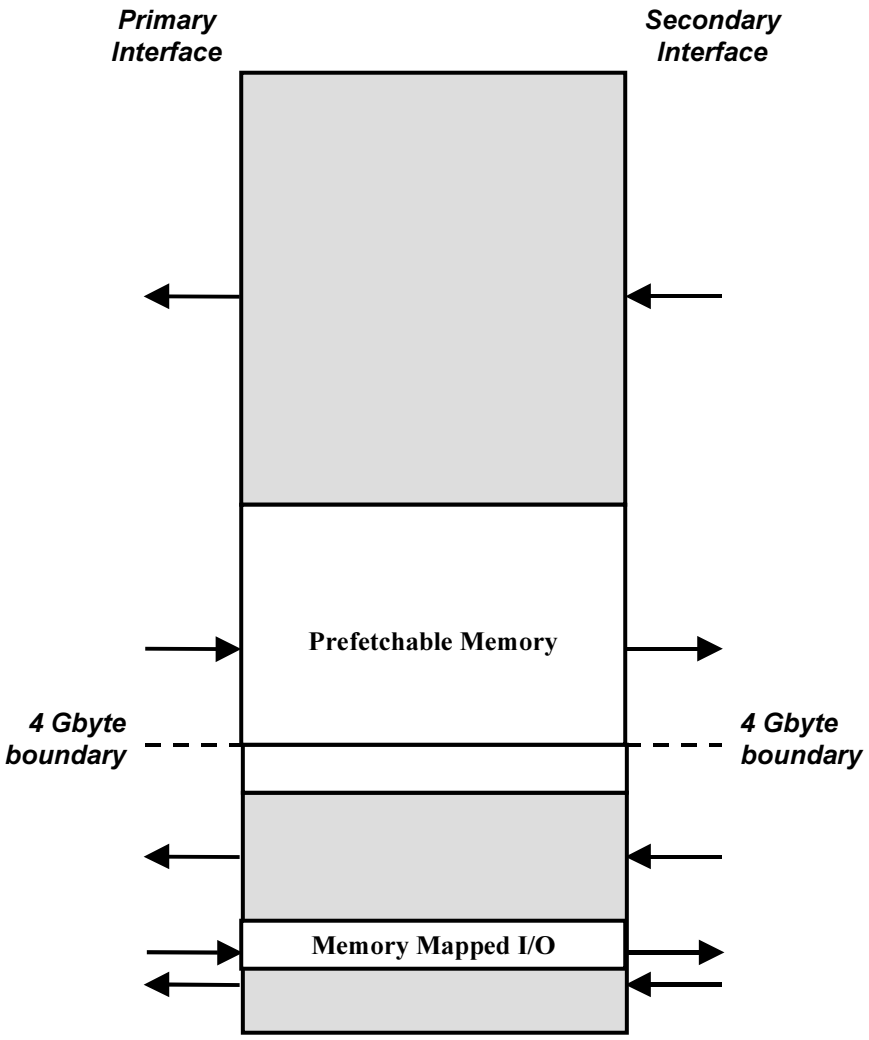



**Figure 4-4: 64-bit Prefetchable Memory Address Range Example**

## 4.4.2. 64-bit Address Decoding of Prefetchable Memory

If a bridge implements 64-bit addressing for prefetchable memory on the secondary interface of the bridge (the Prefetchable Base Upper 32-bits and Prefetchable Limit Upper 32-bits registers are implemented), three different mappings of the prefetchable memory address range are possible.

1. The entire prefetchable memory range resides below the 4 GB address boundary.
2. The entire prefetchable memory range resides above the 4 GB address boundary.
3. The prefetchable memory range straddles the 4 GB boundary.

The 64-bit address decoding behavior of the bridge for each of these possible mappings is described in the following sections.

### 4.4.2.1. Below the 4 GB Boundary

If the entire prefetchable memory address range on the secondary interface resides below the 4 GB address boundary, the Prefetchable Base Upper 32-bits and Prefetchable Limit Upper 32-bits registers must be programmed to be zero. The bridge will ignore all DAC commands on the primary interface, since there are no 64-bit addresses behind the bridge. The address of single address cycles are checked to see if they are equal to or greater than the base address specified in the Prefetchable Memory Base Address register (the low 32-bits of the base address) and less than or equal to the limit address specified in the Prefetchable Memory Limit Address register (the low 32-bits of the limit address). If both address comparisons are true, the bridge claims the access and passes it to the secondary bus. In this case, the Prefetchable Base Upper 32-bits and Prefetchable Limit Upper 32-bits registers are not used in the decode.

### 4.4.2.2. Above the 4 GB Boundary

If the entire prefetchable memory address range on the secondary interface resides above the 4 GB address boundary, the Prefetchable Base Upper 32-bits and Prefetchable Limit Upper 32-bits registers must be programmed with non-zero values. Therefore, the bridge will ignore SAC transactions and only decode DAC commands initiated on the primary interface that are in the prefetchable range, since the range of interest cannot be accessed with a SAC. The addresses of DAC commands are checked to see if they are equal to or greater than the 64-bit prefetchable base address and less than or equal to the 64-bit prefetchable limit address. The 64-bit prefetchable base address is specified by the Prefetchable Base Upper 32-bits register and the Prefetchable Memory Base register. The 64-bit prefetchable limit address is specified by the Prefetchable Limit Upper 32-bits register and the Prefetchable Memory Limit register. If both address comparisons are true, the bridge claims the access and passes it to the secondary bus.

### 4.4.2.3. Across the 4 GB Boundary

If the prefetchable memory address range on the secondary interface straddles the 4 GB address boundary, the Prefetchable Base Upper 32-bits register must be programmed to zero and Prefetchable Limit Upper 32-bits registers must be programmed to a non-zero value. The bridge must decode the address of both SAC and DAC transactions to determine if the access is to be claimed and forwarded to the secondary bus.

The address of a SAC transaction is compared only to the Prefetchable Memory Base register, if the Prefetchable Base Upper 32-bits register is zero. If the 32-bit address of a SAC transaction is equal to or greater than the Prefetchable Memory Base register, the access is claimed by the bridge and forwarded to the secondary bus.

The first address of a DAC transaction is compared to the Prefetchable Memory Limit register and the second address of a DAC transaction is compared to the Prefetchable Limit Upper 32-bits register. If the 64-bit address is less than or equal to the 64-bit prefetchable memory limit address, the access is claimed by the bridge and forwarded to the secondary bus.

## 4.5. VGA Support

There are two issues related to the support of VGA compatible devices in systems with bridges: VGA compatible addressing and VGA palette snooping. Bridges are not required to implement the VGA support mechanisms described in the following sections. However, if a bridge implements the support mechanisms for VGA compatible addressing, it must also implement the mechanisms for VGA palette snooping and vice versa.

If VGA support is implemented by a bridge, the bridge must have the capability to be configured to recognize the ISA compatible addresses used by VGA devices, so it can support a VGA device downstream of the bridge. A bridge must also support configurations where a graphics device downstream of a bridge needs to snoop VGA palette accesses if VGA support is implemented.

### 4.5.1. VGA Compatible Addressing

The VGA Enable bit in the Bridge Control register is used to control response by the bridge to both the VGA frame buffer addresses and to the VGA register addresses. If a VGA compatible device is located downstream of a bridge, the VGA Enable bit must be set. If the VGA Enable bit is set, the bridge will positively decode and forward memory accesses to VGA frame buffer addresses and I/O accesses to VGA registers from the primary interface to the secondary interface (and block forwarding from the secondary to primary interface of these same accesses). A bridge never forwards transactions that access VGA BIOS memory addresses (regardless of the setting of the VGA Enable bit). ROM code provided by PCI compatible devices must be copied to system memory before execution and may be mapped to any address in PCI memory address space via the Expansion ROM Base Address register in the device's configuration header.

VGA memory addresses

- 0A 0000h through 0B FFFFh

VGA I/O addresses (including ISA aliases - **AD[15::10]** are not decoded)

- **AD[9::0]** = 3B0h through 3BBh, and 3C0h through 3DFh

### 4.5.2. VGA Palette Snooping

The behavior of each display device and each bridge in the path of a palette access must be selected based on the configuration of the system. The palette snooping behavior of bridges is described below. For more tutorial information, see Section 12.1.2.

A bridge that implements VGA support must provide the following three modes of VGA palette access:

- Ignore VGA palette accesses

  if there are no graphics agents downstream that need to snoop or respond to VGA palette access cycles (reads or writes)

- Positively decode and forward VGA palette writes

  if there are graphics agents downstream of the bridge that need to respond or snoop palette writes (reads are ignored)

- Positively decode and forward VGA palette reads and writes

  if there are VGA compatible graphics agents that are downstream of the bridge

The VGA Enable bit in the Bridge Control register and the VGA Snoop Enable bit in the Command register select the bridge's response to palette accesses as shown in Table 4-1.

**Table 4-1: Response to VGA Palette Accesses**

| VGA Enable | VGA Snoop Enable | Bridge Response to Palette Accesses |
|---|---|---|
| 0 | 0 | Ignore all palette accesses |
| 0 | 1 | Positively decode palette writes (ignore reads) |
| 1 | x | Positively decode palette reads and writes |

The VGA palette addresses are as follows (inclusive of ISA aliases - **AD[15::10]** are not decoded):

- **AD[9::0]** = 3C6h, 3C8h, and 3C9h

## 4.6. Subtractive Decode Support

A unique class code of 060401h has been assigned to bridges that support subtractive decoding. See the *PCI Local Bus Specification* for the device selection requirements of a subtractive decoding device. The purpose of the subtractive decoding class code is to let configuration software know the bridge supports subtractive decoding and, therefore, may have subtractive decoding devices on the secondary bus. The only unique feature indicated by class code 060401h is the support of subtractive decoding in addition to all the other requirements of this specification. No other functional characteristics should be assumed of bridges that have a 060401h class code. A complete description of system features associated with a subtractive decoding bridge is beyond the scope of this specification.

# Chapter 5
# Buffer Management

## 5.1. Prefetching Read Data

The term prefetch is used when the bridge reads data from the target in anticipation that the master will consume it. Prefetching is a useful technique for hiding the latency of a burst read transaction but its use is restricted. Memory that is prefetchable has the attribute that it returns the same data when read multiple times and does not alter any other device state when it is read[3]. When prefetching, the bridge may read data that is not consumed by the master. The bridge is required to discard any prefetched read data not consumed when the master concludes the read transaction (refer to Section 5.6.2.).

The *PCI Local Bus Specification* specifies that a bridge may safely prefetch data when the transaction uses the Memory Read Line or Memory Read Multiple command. Since most processor architectures do not have the notion of prefetchable memory, typical host bus bridges do not generate Memory Read Line or Memory Read Multiple transactions. A bridge may provide an optional address range that allows the bridge to prefetch memory read data from a target attached to the secondary interface of the bridge. A target explicitly indicates to configuration software that a memory address range is prefetchable by setting the Prefetchable bit (bit 3) in the corresponding Base Address Register (BAR) within the target's Configuration Space header. Configuration software uses this information to program the prefetchable memory address range in the bridge and to map the prefetchable address ranges of secondary bus targets into the range. If a master on the primary interface of a bridge accesses a location mapped in the prefetchable memory address range, the bridge is permitted to prefetch read data when completing the transaction on the secondary bus. In this case, the bridge is permitted either to extend the read transaction burst length or to modify the bus command to Memory Read Line or Memory Read Multiple or both. This function is supported only if the optional Prefetchable Memory Base and Limit registers are implemented.

A bridge is also permitted to prefetch data from a secondary bus if the transaction originates on the primary bus and is either a Memory Read Line or Memory Read Multiple command. A

[3] In a prior revision of this specification, a PCI-PCI bridge was not permitted to prefetch read data across a 4 KB boundary. However, this restriction has been removed by this specification revision. It is the responsibility of the target device to disconnect a burst transaction when either a BAR boundary is reached, or a boundary is reached within a BAR where the attributes of the access change (i.e., prefetchable vs. non-prefetchable).

bridge is permitted to prefetch data from the primary bus if the transaction originates on the secondary bus and is either a Memory Read Line or Memory Read Multiple command. Masters attached to the secondary interface of a PCI-to-PCI bridge are encouraged to use the Memory Read Line or Memory Read Multiple commands if they desire high performance (prefetchable) read transactions. The bridge is also permitted to assume that all transactions that originate on the secondary bus and go up through the bridge have a final destination at main memory and therefore are prefetchable. If a bridge makes this assumption and does blind prefetching on the Memory Read command, it must support a device-specific bit (in Configuration Space) that allows this feature to be disabled (if blind prefetching causes a problem). When prefetching memory read data, the bridge is permitted to assert all byte enables for all data phases on the destination bus independent of the byte enables used by the originating bus master.

Table 5-1 lists when prefetching by the bridge is permitted for those bus commands that function like a read transaction (i.e., the master is reading data from the target device). Prefetching does not apply for those bus commands where the master writes data to the target device, the Dual Address Command encoding, or for reserved bus command encodings.

**Table 5-1: Read Prefetch Summary**

| C/BE[3::0]# | Command type | Access Originates on the: | |
|---|---|---|---|
| | | Primary Bus | Secondary Bus |
| | | | |
| 0010 | I/O Read | No | No |
| 0110 | Memory Read | No (Note 1) | No (Note 2) |
| 1010 | Configuration Read | No | No |
| 1100 | Memory Read Multiple | Yes | Yes |
| 1110 | Memory Read Line | Yes | Yes |

Notes:

1. Yes when the address is in the prefetchable range as described by Prefetchable Memory Base and Limit registers.
2. The bridge is permitted to treat this like Memory Read Line or Memory Read Multiple, but this feature must be able to be turned off via a device-specific bit.

Prefetching of read data is never allowed in the PCI I/O Space or Configuration Space.

## 5.2. Posting Write Data

Posting of write data is required by the *PCI Local Bus Specification* if either Memory Write or Memory Write and Invalidate commands are used for transactions that cross the bridge in either direction and the bridge has posting buffer space available. Posting of I/O Write and Configuration Write transactions is not permitted by a bridge.

A PCI-to-PCI bridge is generally allowed to terminate with Retry a transaction that uses the Memory Write or Memory Write and Invalidate commands only when its buffers are filled with previously received memory write data or for a locked operation. Terminating a Memory Write or Memory Write and Invalidate transaction with Retry for other reasons can lead to deadlocks. Refer to Section 5.6.3. for more details.

### 5.2.1. Memory Write and Invalidate Usage

The *PCI Local Bus Specification* permits a master to use the Memory Write and Invalidate (MWI) command to transfer memory write data when the following requirements are met:

- linear incrementing address mode is used
- the transaction begins on a cacheline aligned boundary
- the master guarantees that it will deliver all bytes within any cacheline accessed (in some cases the transaction will be a burst transaction which accesses multiple cachelines)

When functioning as a bus master a bridge is permitted to use the MWI command when forwarding transactions (upstream or downstream) in one of three cases:

- the bridge forwards an MWI transaction originated by another master
- the bridge promotes a Memory Write (MW) transaction (or portion of a MW transaction) to a MWI transaction
- the bridge combines sequential MW transactions to generate a transaction that meets the MWI usage requirements

Note that for each case there are requirements that qualify the bridges ability to use the MWI command for the transaction. Each case is described in detail in the following sections.

#### 5.2.1.1. Forwarding Memory Write and Invalidate Transactions

A bridge is permitted to forward a MWI transaction originated by another master to the opposite interface as a MWI transaction when the following conditions are true:

- the bridge implements the Cacheline Size register (see Section 3.2.4.7.)
- the Cacheline Size register has been set to a value supported by the bridge

The bridge is not required to validate that the forwarded transaction meets the MWI usage requirements (the originating master is responsible for meeting the MWI usage requirements). The setting of the Memory Write and Invalidate bit in the Command register (see Section 3.2.4.3.) does

not affect the bridge's ability to use the MWI command on the destination bus when forwarding a MWI transaction originated by another master.

If the bridge does not implement the Cacheline Size register, or the Cacheline Size is not set to a value supported by the bridge, then the bridge must change the MWI command to MW when forwarding the transaction across the bridge[4].

### 5.2.1.2. Promoting Memory Write Transactions

When forwarding a MW transaction, the bridge can optionally promote the transaction to a MWI transaction if it meets the MWI usage rules. In this case, the bridge is required to validate that the forwarded transaction meets all MWI usage requirements. To use this method, the bridge must implement the Memory Write and Invalidate bit in the Command register (see Section 3.2.4.3.) and the Cacheline Size register (see Section 3.2.4.7.). Additionally, the Memory Write and Invalidate bit must be set to enable the bridge promote Memory Write transactions to MWI transactions.

A bridge is permitted to meet the MWI usage rules by promoting only a subset of a MW transaction. For example, consider a MW burst transaction that begins and ends on address boundaries that are not cacheline aligned but that contain one or more cachelines within the burst that have all bytes enabled. When forwarding the MW transaction, the bridge is permitted to meet the MWI usage requirements by segmenting the original MW transaction into three separate transactions on the destination bus. The first transaction would use a MW transaction to transfer the memory write data from the starting address (which is not cacheline aligned) up to the next aligned cacheline boundary. The bridge would then use a MWI transaction to transfer the memory write data beginning on the aligned cacheline boundary including all subsequent complete cachelines up to the final aligned cacheline boundary contained in the original MW transaction. The bridge would then use a MW transaction to transfer the remainder of the data contained in the original MW transaction. Note that the bridge cannot alter the ordering of the original MW transaction. All bytes must be forwarded in the order they were received in the original MW transaction.

### 5.2.1.3. Combining Memory Write Transactions

A bridge may optionally combine sequential Memory Write transactions (see Section 5.7.) to generate a transaction that meets MWI usage requirements. In this case, the bridge is required to validate that the forwarded transaction meets all MWI usage requirements. To use this method, the bridge must implement the Memory Write and Invalidate bit in the Command register (see Section 3.2.4.3.) and the Cacheline Size register (see Section 3.2.4.7.). Additionally, the Memory Write and Invalidate bit must be set to enable the bridge to combine Memory Write transactions into MWI transactions. Note that the bridge cannot alter the ordering of the original MW transaction. All bytes must be forwarded in the order they were received in the original MW transaction.

---

[4] A valid Cacheline Size is necessary for the bridge to determine cacheline boundaries on the target bus when the Master Latency Timer on the destination bus expires during the delivery of a MWI transaction (see Sections 3.2.4.8. and 3.2.5.5.).

## 5.2.1.4. Memory Write and Invalidate Disconnects

The *PCI Local Bus Specification* permits a target to disconnect a Memory Write and Invalidate transaction on an address that is not an aligned cacheline boundary. In this case, when the master resumes delivery of the remaining write data for the cacheline in which the disconnect occurred, the master must use a Memory Write transaction (the Memory Write and Invalidate usage requirements are no longer valid). The following sections describe the requirements for delivering the remainder of a MWI transaction for two cases of Target-Disconnect:

- the bridge disconnects the originating master on the originating bus
- the target disconnects the bridge on the destination

These two cases are discussed in the following sections.

### 5.2.1.4.1. Master Disconnected by the Bridge

When responding as a target, a bridge can disconnect a master during any data phase of a MWI transaction (this may occur as the result of a temporary buffer full condition for example). The bridge must use a MW transaction on the destination bus to deliver the write data for the incomplete cacheline in which the disconnect occurred. The bridge is permitted to use the MWI command to forward any cachelines posted by the bridge *prior* to signaling disconnect to the originating master provided the requirements detailed in Section 5.2.1.1. are met.

### 5.2.1.4.2. Bridge Disconnected by the Target

When a bridge is disconnect by the target when forwarding a MWI transaction on an address boundary that is not cacheline aligned, the bridge must use a MW transaction to deliver the remaining write data for the cacheline in which the disconnect occurred. The bridge is permitted to use the techniques described in Sections 5.2.1.2. and 5.2.1.3. to resume delivery of subsequent cachelines with MWI transactions.

## 5.3. Delayed Transactions

The following discussion of Delayed Transactions is included to clarify their application to PCI-to-PCI bridges. For a complete treatment of Delayed Transactions, refer to the *PCI Local Bus Specification*.

Bridges must implement Delayed Transactions to meet the latency requirements of the *PCI Local Bus Specification*. Delayed Transactions significantly improve the transfer efficiency of PCI buses with as few as one bridge since the master is not held in wait states while the bridge completes the transaction on the destination bus. Furthermore, Delayed Transactions allow more combinations of transactions crossing the bridge in opposite directions to run concurrently than would otherwise be possible thus avoiding potential starvation problems and improving efficiency.

Only non-posted transactions can be completed as Delayed Transactions by a bridge. These include I/O Read, I/O Write, Configuration Read, Configuration Write, Memory Read, Memory Read Line, and Memory Read Multiple. Memory Write and Memory Write and Invalidate transactions are postable and, therefore, cannot be completed as Delayed Transactions.

To complete a transaction using Delayed Transaction termination, a bridge must latch the following information:

- address
- command
- byte enables
- address and data parity
- **REQ64#** (if a 64-bit transfer)

For write transactions completed using Delayed Transaction termination, a bridge must also latch data from byte lanes for which the byte enable is asserted and may optionally latch data from byte lanes for which the byte enable is deasserted. **LOCK#** must also be latched for transactions flowing downstream. Refer to Section 5.4. for additional requirements when completing a Delayed Transaction as part of a locked operation.

After latching the required information, the bridge terminates the transaction on the originating bus with Retry. Once ordering requirements have been satisfied (see Section 5.5.), and the arbiter has granted the destination bus to the bridge, the bridge begins the process of executing the transaction on the destination bus. If the Delayed Request is a read, the bridge obtains the requested data and completion status. If the Delayed Request is a write, the bridge delivers the write data and obtains the completion status. Completing the Delayed Request on the destination bus produces a Delayed Completion which consists of the latched information of the Delayed Request and the completion status (and data if a read request). The bridge stores the Delayed Completion until the master repeats the initial request.

The bridge differentiates between transactions (by the same or different masters) by comparing the current transaction with information latched previously (for both Delayed Requests and Delayed Completions). When the Parity Error Response bit (bit 6 of the Command Register for the primary bus and bit 0 of the Bridge Control register for the secondary bus) is cleared, the

bridge ignores the address and data parity latched previously when doing the comparison. The byte enables may optionally be ignored in the comparison if the master is reading from a prefetchable location, even though the master is required to repeat the transaction with the same byte enables. If the compare matches a Delayed Request (already enqueued), but the bridge is not ready to complete the request, the bridge does not enqueue the request again but simply terminates the transaction with Retry. If the compare matches a Delayed Completion and the bridge is ready to complete the request, the bridge responds by signaling the status and provides the data if a read transaction.

The master must repeat the transaction exactly as the original request; otherwise, the bridge will assume it is a new transaction (since the agent cannot distinguish masters). Two masters could request the exact same transaction and the bridge cannot and need not distinguish between them and will simply complete the Delayed Transaction with the first master to repeat the transaction after the completion on the destination bus by the bridge.

A bridge is permitted to enqueue one or more Delayed Requests at a time. If a bridge enqueues multiple Delayed Requests, the order in which it attempts them on the destination bus is independent of the order in which they were originally attempted on the originating bus. Furthermore, the order in which the transactions ultimately complete on the originating bus is independent of the order in which they were attempted on either bus and the order in which they completed on the secondary bus. (Refer to Section 5.5., for restrictions with respect to posted memory writes.)

While completing a Delayed Request, the bridge may, from time to time, terminate a memory write transaction with Retry while temporary internal conflicts are being resolved; for example, when all the memory-write data buffers are full or during a locked transaction. However, the bridge cannot require a Delayed Transaction to complete on the originating bus before accepting the memory write data from a master on that bus; otherwise, a deadlock may occur. Furthermore, the bridge cannot indefinitely terminate a memory write with Retry on one bus because it is waiting to run a previously enqueued Delayed Request on the other bus, or because it is waiting for a master to take a previously enqueued Delayed Completion on either bus. Refer to Section 5.5. and Section 5.6.3. for more details.

## 5.3.1. Discarding a Delayed Request

Since a Delayed Request is only a request and not really a transaction, the bridge is allowed to discard a Delayed Request from the time it is enqueued until it has been attempted on the destination bus. Once a request has been attempted on the destination bus, it must continue to be repeated until it completes on the destination bus and cannot be discarded. The bridge is allowed to present other requests. But if it attempts more than one request, the bridge must continue to repeat all requests that have been attempted unconditionally until they complete. The repeating of the requests is not required to be equal, but is required to be fair.

The bridge is allowed to discard Delayed Completions in only two cases. The first case is if the Delayed Completion is a read of a prefetchable region (or the command was Memory Read Line or Memory Read Multiple). The second case is for all Delayed Completions (read or write, prefetchable or not) if the master has not repeated the request before the Discard Timer interval is exceeded. When this occurs, the device is required to discard the Delayed Completion; otherwise, a deadlock may occur.

### 5.3.2. Discarding a Delayed Completion

The Discard Timer for masters on the primary bus is selectable to expire either within $2^{15}$ clocks or $2^{10}$ clocks depending on the state of bit 8 in the Bridge Control register. The Discard Timer for masters on the secondary bus is similarly controlled by bit 9 in the Bridge Control register. The longer value is the default and should be adequate for most systems. However, some classes of devices designed before Delayed Transactions were introduced into the *PCI Local Bus Specification* may encounter situations in which a transaction terminated with Retry is not repeated. If this situation occurs frequently enough that the longer Discard Timer value causes a performance problem, then the shorter time can be selected. When the Discard Timer interval is exceeded on either the primary interface or the secondary interface, the bridge must set the Discard Timer Status bit (bit 10) in the Bridge Control register. In addition, the bridge must assert **SERR#** on the primary interface if enabled to do so by the Discard Timer SERR# Enable bit (bit 11) in the Bridge Control register and the SERR# Enable bit in the Command register.

While completing a Delayed Request, the bridge may, from time to time, terminate a memory write transaction with Retry while temporary internal conflicts are being resolved; for example, when all the memory-write data buffers are full or during a locked transaction. However, the bridge cannot require a Delayed Transaction to complete on the originating bus before accepting the memory write data from a master on that bus; otherwise, a deadlock may occur. Furthermore, the bridge cannot indefinitely terminate a memory write with Retry on one bus because it is waiting to run a previously enqueued Delayed Request on the other bus, or because it is waiting for a master to take a previously enqueued Delayed Completion on either bus. Refer to Section 5.5. and Section 5.6.3. for more details.

## 5.4. Exclusive Access Transactions

A PCI-to-PCI bridge is only allowed to propagate an exclusive access transaction from its primary to its secondary interface and never allowed to initiate an exclusive access of its own initiative. A PCI-to-PCI bridge is required to ignore **LOCK#** when acting as a target on its secondary interface. A PCI-to-PCI bridge adheres to the **LOCK#** usage requirements defined in the *PCI Local Bus Specification*. This section describes additional requirements for propagating an exclusive access across a PCI-to-PCI bridge.

The first transaction of a lock operation must be a read transaction and is completed by the bridge using Delayed Transaction termination. When a downstream read transaction with **LOCK#** is successfully queued as a Delayed Request (Delayed Lock-Request) on the primary interface, the bridge enters a *target-lock* state even though the master is terminated with Retry. While in the target-lock state, the bridge terminates with Retry all transactions on the primary interface.

When the downstream Delayed Lock-Request is queued on the primary interface, it is forwarded to the secondary interface exactly the same as an unlocked Delayed Request (i.e., all ordering rules are obeyed. See Section 5.5.). The bridge must follow the requirements for initiating an exclusive access before attempting the Delayed Lock-Request.

### 5.4.1. Delayed Lock-Request Error

If the Delayed Lock-Request completes on the secondary interface with Master-Abort or Target-Abort, the bridge has not successfully established exclusive access of the target device and does not establish control of the secondary interface **LOCK#** resource. In this case, the bridge is allowed to respond to subsequent upstream transactions normally (i.e., does not unconditionally terminate with Retry upstream posted writes or upstream Delayed Transactions). When the Delayed Lock-Request terminates with Master-Abort or Target-Abort on the secondary interface, the error information is retained in the Delayed Lock-Completion that is produced.

The Delayed Lock-Completion is returned to the primary interface exactly the same as an unlocked Delayed Completion (i.e., all ordering rules are obeyed; refer to Section 5.5.). When the Delayed Lock-Completion is returned to the originating master, the bridge is required to signal a Target-Abort to the originating master on the primary interface. When the bridge signals Target-Abort to the originating master, the primary interface of the bridge transitions from the target-lock state to an unlocked state (lock is not established). In this case, the master has not successfully established exclusive access of the target device, and it does not establish control of the primary interface **LOCK#** resource.

### 5.4.2. Normal Completion

If the Delayed Lock-Request completes without error on the secondary interface (normal completion or disconnect), then the bridge has successfully established exclusive access of the target device and ownership of the secondary interface **LOCK#** resource. The secondary interface of the bridge enters a locked state, and the bridge must terminate with Retry all subsequent upstream posted writes. The bridge may optionally terminate with Retry upstream Delayed Transactions while the secondary interface is in the locked state.

The Delayed Lock-Completion is returned to the primary interface exactly the same as an unlocked Delayed Completion (i.e., all ordering rules are obeyed). When the Delayed Lock-Completion is returned to the originating master, the bridge signals normal completion (including disconnect). At this point, the primary interface transitions from the target-lock state to a full-lock state (lock is established across the bridge). In this case, the master has successfully established exclusive access of the target device and ownership of the primary interface **LOCK#** resource.

Once **LOCK#** is established between the originating master and the target device, both the primary and secondary interfaces of the bridge remain in their locked states. Both interfaces remain in the locked state until the originating master releases **LOCK#** ownership on the primary interface. The originating master relinquishes **LOCK#** ownership when it completes the desired sequence of exclusive operations or the bridge signals a Target-Abort to the master. When lock ownership is relinquished by the originating master, the primary interface of the bridge changes from the locked state to the unlocked state, the bridge relinquishes **LOCK#** ownership on the secondary bus, and the secondary interface of the bridge changes from the locked state to the unlocked state.

## 5.5. Ordering Requirements

Appendix E of the *PCI Local Bus Specification* specifies the ordering requirements for PCI-PCI bridges. Sections of Appendix E are duplicated here for convenience. For a full discussion of PCI ordering requirements, refer to the *PCI Local Bus Specification*.

## Summary of PCI Ordering Requirements

Following is a summary of the general PCI ordering requirements presented in the *PCI Local Bus Specification*.

**General Requirements**

1. The order of a transaction is determined when it completes. Transactions terminated with Retry are only requests and can be handled by the bridge in any order.
2. Memory writes can be posted in both directions in a bridge. I/O and Configuration writes are not posted. (I/O writes can be posted in the host bridge, but some restrictions apply.) Read transactions (Memory, I/O, or Configuration) are not posted.
3. Posted memory writes moving in the same direction through a bridge will complete on the destination bus in the same order they complete on the originating bus.
4. Write transactions crossing a bridge in opposite directions have no ordering relationship.
5. A read transaction must push ahead of it through the bridge any posted writes originating on the same side of the bridge and posted before the read. Before the read transaction can complete on its originating bus, it must pull out of the bridge any posted writes that originated on the opposite side and were posted before the read command completes on the read-destination bus.
6. A bridge can never make the acceptance (posting) of a memory write transaction as a target contingent on the prior completion of a non-locked transaction as a master on the same bus. Otherwise, a deadlock may occur. Bridges are allowed to refuse to accept a memory write for temporary conditions which are guaranteed to be resolved with time. A bridge can make the acceptance of a memory write transaction as a target contingent on the prior completion of locked transaction as a master only if the bridge has already established a locked operation with its intended target.

The following is a summary of the PCI ordering requirements specific to Delayed Transactions presented in the *PCI Local Bus Specification*.

**Delayed Transaction Requirements**

1. A target that uses Delayed Transactions may be designed to have any number of Delayed Transactions outstanding at one time.
2. Only non-posted transactions can be handled as Delayed Transactions.
3. A master must repeat any transaction terminated with Retry since the target may be using a Delayed Transaction.

4. Once a Delayed Request has been attempted on the destination bus, it must continue to be repeated until it completes on the destination bus. Before it is attempted on the destination bus, it is only a request and may be discarded at anytime.
5. A Delayed Completion can only be discarded when it is a read from a prefetchable region, or if the master has not repeated the transaction in $2^{15}$ or $2^{10}$ clocks.
6. A target must accept all memory writes addressed to it even while completing a request using Delayed Transaction termination.
7. Delayed Requests and Delayed Completions have no ordering requirements with respect to themselves.
8. Delayed Completions must be given an opportunity to pass Delayed Requests.
9. Only a Delayed Write Completion can pass a Posted Memory Write. A Posted Memory Write must be given an opportunity to pass everything except another Posted Memory Write.
10. A single master may have any number of outstanding requests terminated with Retry. However, if a master requires one transaction to be completed before another, it cannot attempt the second one on PCI until the first one has completed.

## Ordering of Requests

A transaction is considered to be a *request* when it is presented on the bus. When the transaction is terminated with Retry, it is still considered a request. A transaction becomes *complete* or a *completion* when data actually transfers (or is terminated with Master-Abort or Target-Abort). The following discussion will refer to a transaction as being a request or completion depending on the success of the transaction.

A transaction that is terminated with Retry has no ordering relationship with any other access. Ordering of accesses is only determined when an access completes (transfers data). For example, four masters A, B, C, and D reside on the same bus segment and all desire to generate an access on the bus. For this example, each agent can only request a single transaction at a time and will not request another until the current access completes. The order in which transactions complete are based on the algorithm of the arbiter and the response of the target, not the order in which each agent's **REQ#** signal was asserted. Assuming that some requests are terminated with Retry, the order in which they complete is independent of the order they were first requested. By changing the arbiter's algorithm, the completion of the transactions can be any sequence (i.e., A, B, C, and then D or B, D, C, and then A, and so on). Because the arbiter can change the order in which transactions are requested on the bus, and, therefore, the completion of such transactions, the system is allowed to complete them in any order it desires. This means that a request from any agent has no relationship with a request from any other agent. The only exception to this rule is when **LOCK#** is used, which is described later.

Take the same four masters (A, B, C, and D) used in the previous paragraph and integrate them onto a single piece of silicon (a multi-function device). For a multi-function device, the four masters operate independent of each other, and each function only presents a single request on the bus for this discussion. The order their requests complete is the same as if they where separate agents and not a multi-function device, which is based on the arbitration algorithm. Therefore, multiple requests from a single agent may complete in any order, since they have no relationship to each other.

Another device, not a multi-function device, has multiple internal resources that can generate transactions on the bus. If these different sources have some ordering relationship, then the device must ensure that only a single request is presented on the bus at any one time. The agent must not attempt a subsequent transaction until the previous transaction completes. For example, a device has two transactions to complete on the bus, Transaction A and Transaction B and A must complete before B to preserve internal ordering requirements. In this case, the master cannot attempt B until A has completed.

The following example would produce inconsistent results if it were allowed to occur. Transaction A is to a flag that covers data, and Transaction B accesses the actual data covered by the flag. Transaction A is terminated with Retry, because the addressed target is currently busy or resides behind a bridge. Transaction B is to a target that is ready and will complete the request immediately. Consider what happens when these two transactions are allowed to complete in the wrong order. If the master allows Transaction B to be presented on the bus after Transaction A was terminated with Retry, Transaction B can complete before Transaction A. In this case, the data may be accessed before it is actually valid. The responsibility to prevent this from occurring rests with the master, which must block Transaction B from being attempted on the bus until Transaction A completes. A master presenting multiple transactions on the bus must ensure that subsequent requests (that have some relationship to a previous request) are not presented on the bus until the previous request has completed. The system is allowed to complete multiple requests from the same agent in any order. When a master allows multiple requests to be presented on the bus without completing, it must repeat each request independent of how any of the other requests complete.

## Ordering of Delayed Transactions

A Delayed Transaction progresses to completion in three phases:

1. Request by the master
2. Completion of the request by the target
3. Completion of the transaction by the master

During the first phase, the master generates a transaction on the bus, the target decodes the access, latches the information required to complete the access, and terminates the request with Retry. The latched request information is referred to as a Delayed Request. During the second phase, the target independently completes the request on the destination bus using the latched information from the Delayed Request. The result of completing the Delayed Request on the destination bus produces a Delayed Completion which consists of the latched information of the Delayed Request and the completion status (and data if a read request). During the third phase, the master successfully re-arbitrates for the bus and reissues the original request. The target decodes the request and gives the master the completion status (and data if a read request). At this point, the Delayed Completion is retired and the transaction has completed.

The number of simultaneous Delayed Transactions a bridge is capable of handling is limited by the implementation and not by the architecture. Table 5-2 represents the ordering rules when a bridge in the system is capable of allowing multiple transactions to proceed in each direction at the same time. Each column of the table represents an access that was accepted by the bridge earlier, while each row represents a transaction just accepted. The contents of the box indicate what ordering relationship the second transaction must have to the first.

**PMW** - *Posted Memory Write* is a transaction that has completed on the originating bus before completing on the destination bus and can only occur for Memory Write and Memory Write and Invalidate commands.

**DRR** - *Delayed Read Request* is a transaction that must complete on the destination bus before completing on the originating bus and can be an I/O Read, Configuration Read, Memory Read, Memory Read Line, or Memory Read Multiple commands. As mentioned earlier, once a request has been attempted on the destination bus, it must continue to be repeated until it completes on the destination bus. Before it is attempted on the destination bus, the DRR is only a request and may be discarded at any time to prevent deadlock or improve performance since the master must repeat the request later.

**DWR** - *Delayed Write Request* is a transaction that must complete on the destination bus before completing on the originating bus and can be an I/O Write or Configuration Write commands. Note: Memory Write and Memory Write and Invalidate transactions must be posted (PMW) and not be completed as DWR. As mentioned earlier, once a request has been attempted on the destination bus, it must continue to be repeated until it completes. Before it is attempted on the destination bus, the DWR is only a request and may be discarded at any time to prevent deadlock or improve performance since the master must repeat the request later.

**DRC** - *Delayed Read Completion* is a transaction that has completed on the destination bus and is now moving toward the originating bus to complete. The DRC contains the data requested by the master and the completion status (normal, Master-Abort, Target-Abort, parity error, etc.).

**DWC** - *Delayed Write Completion* is a transaction that has completed on the destination bus and is now moving toward the originating bus. The DWC does not contain the data of the access but only status of how it completed (Normal, Master-Abort, Target-Abort, parity error, etc.). The write data has been written to the specified target.

**No** - indicates that the subsequent transaction is not allowed to complete before the previous transaction to preserve ordering in the system. The four No boxes found in column 2 prevent PMW data from being passed by other accesses and thereby maintain a consistent view of data in the system.

**Yes** - indicates that the subsequent transaction must be allowed to complete before the previous one or a deadlock can occur.

When blocking occurs, the PMW is required to pass the DRC or the DWC. If the master continues attempting to complete Delayed Requests, it must be fair in attempting to complete the PMW. [DN]

**Yes/No** - indicates that the bridge designer may choose to allow the subsequent transaction to complete before the previous transaction or not. This is allowed since there are no ordering requirements to meet or deadlocks to avoid. How a bridge designer chooses to implement these boxes may have a cost impact on the bridge implementation or performance impact on the system.

**Table 5-2: Ordering Rules for a Bridge**

| **Row pass Col.?** | **PMW** (Col 2) | **DRR** (Col 3) | **DWR** (Col 4) | **DRC** (Col 5) | **DWC** (Col 6) |
|---|---|---|---|---|---|
| **PMW** (Row 1) | No[1] | Yes[5] | Yes[5] | Yes[7] | Yes[7] |
| **DRR** (Row 2) | No[2] | Yes/No | Yes/No | Yes/No | Yes/No |
| **DWR** (Row 3) | No[3] | Yes/No | Yes/No | Yes/No | Yes/No |
| **DRC** (Row 4) | No[4] | Yes[6] | Yes[6] | Yes/No | Yes/No |
| **DWC** (Row 5) | Yes/No | Yes[6] | Yes[6] | Yes/No | Yes/No |

Rule 1 - A subsequent PMW cannot pass a previously accepted PMW. (Col 2, Row 1)

Posted Memory write transactions must complete in the order they are received. If the subsequent write is to the flag that covers the data, the Consumer may use stale data if write transactions are allowed to pass each other.

Rule 2 - A read transaction must push posted write data to maintain ordering. (Col 2, Row 2)

For example, a memory write to a location and followed by an immediate memory read of the same location returns the new value (refer to the Special Considerations Section of the *PCI Local Bus Specification*, for possible exceptions). Therefore, a memory read cannot pass posted write data. An I/O read cannot pass a PMW, because the read may be ensuring the write data arrives at the final destination.

Rule 3 - A non-postable write transaction must push posted write data to maintain ordering. (Col 2, Row 2)

A Delayed Write Request may be the flag that covers the data previously written (PMW), and, therefore, the flag cannot pass the data that it potentially covers. (Col 2, Row 3)

Rule 4 - A read transaction must pull write data back to the originating bus of the read transaction. (Col 2, Row 4)

For example, the read of a status register of the device writing data to memory must not complete before the data is pulled back to the originating bus. Otherwise, stale data may be used.

Rule 5 - A Posted Memory Write must be allowed to pass a Delayed Request (read or write) to avoid deadlocks. (Col 3 and Col 4, Row 1)

Referring to Figure 5-1, bridge Y (using Delayed Transactions) is between bridges X and Z (designed to a previous version of this specification and not using Delayed Transactions). Consider the following sequence of events:

- Master 1 initiates a read to Target 1 that is forwarded through bridge X and is queued as a Delayed Request in bridge Y.
- Master 3 initiates a read to Target 3 that is forwarded through bridge Z and is queued as a Delayed Request in bridge Y.
- After Masters 1 and 3 are terminated with Retry, Masters 2 and 4 begin long memory write transactions addressing Targets 2 and 4 respectively, which are posted in the write buffers of bridges X and Z respectively.

When bridge Y attempts to complete the read in either direction, bridges X and Z must flush their posted write buffers before allowing the Read Request to pass through it. If the posted write buffers of bridges X and Z are larger than those of bridge Y, bridge Y's buffers will fill. Bridge Y cannot discard the read request since it has been attempted, and it cannot accept any more write data until the read in the opposite direction is completed. Since this condition exists in both directions, neither DRR can complete because the other is blocking the path. Therefore, the PMW data is required to pass the DRR when the DRR blocks forward progress of PMW data.

The same condition exists when a DWR sits at the head of both queues, since some old bridges also require the posting buffers to be flushed on a non-posted write cycle.

Rule 6 – Delayed Completion (read and write) must be allowed to pass Delayed Requests (read or write) to avoid deadlocks. (Cols 3 and 4, Rows 4 and 5)

Consider an application where the common PCI bus segment is on the secondary bus of bridge A and the primary bus for bridge B. If both bridges do not allow Delayed Completions to pass the Delayed Requests, neither can make progress.

For example, suppose bridge A's request to bridge B completes on bridge B's secondary bus, and bridge B's request completes on bridge A's primary bus. Bridge A's completion is now behind bridge B's request and bridge B's completion is behind bridge A's requests. Therefore, Delayed Completions must be allowed to pass Delayed Requests.

Rule 7 - A Posted Memory Write must be allowed to pass a Delayed Completion (read or write) to avoid deadlocks. (Col 5 and Col 6, Row 1)

Consider another transaction scenario similar to that for Rule 5 (again refer to Figure 5-1). In this case, however, a DRC sits at the head of the queues in both directions of bridge Y at the same time. Again the old bridges (X and Z) contain posted write data from another master. The problem in this case, however, is that the read transaction cannot be *repeated* until all the posted write data is flushed out of the old bridge and the master is allowed to repeat its original request. Eventually, the new bridge cannot accept any more posted data because its internal buffers are full, and it cannot drain them until the DRC at the other end completes. When this condition exists in both directions, neither DRC can complete, because the other is blocking the path. Therefore, the PMW data is required to pass the DRC when the DRC blocks forward progress of PMW data.

The same condition exists when a DWC sits at the head of both queues.

**Transactions that have no ordering constraints**

Some Delayed Transactions (enqueued as Delayed Requests or Delayed Completions) have no ordering relationship with any other Delayed Requests or Delayed Completions. For example, the designer can (for performance or cost reasons) allow or disallow Delayed Requests to pass other Delayed Requests and Delayed Completions that were previously enqueued.

Delayed Requests can pass other Delayed Requests (Cols 3 and 4, Rows 2 and 3).

Since Delayed Requests have no ordering relationship with other Delayed Requests, these four boxes are don't cares.

Delayed Requests can pass Delayed Completion (Col 5 and 6, Rows 2 and 3).

Since Delayed Requests have no ordering relationship with Delayed Completions, these four boxes are don't cares.

Delayed Completions can pass other Delayed Completion (Col 5 and 6, Rows 4 and 5).

Since Delayed Completions have no ordering relationship with other Delayed Completions, these four boxes are don't cares. Delayed Write Completions can pass posted memory writes or be blocked by them (Col 2, Row 5)
If the DWC is allowed to pass a PMW or if it remains in the same order, there is no deadlock or data inconsistencies in either case. The DWC data and the PMW data are moving in opposite directions, initiated by masters residing on different buses accessing targets on different buses.

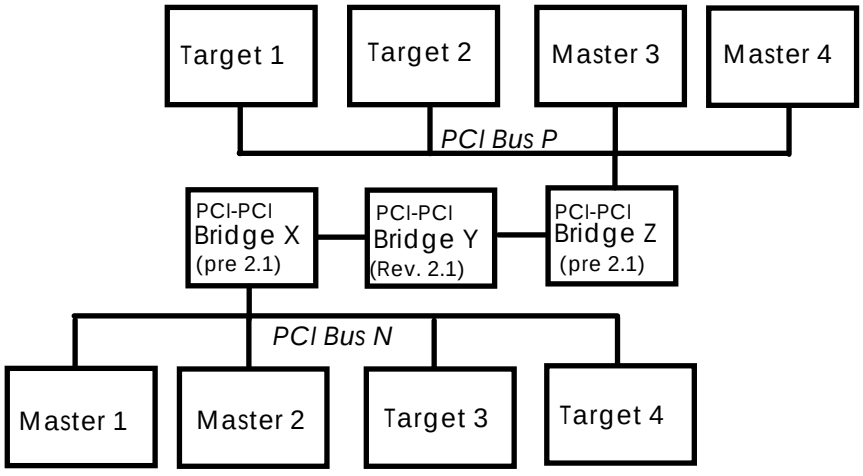



**Figure 5-1: Example System with PCI-to-PCI Bridges**

## Delayed Transactions and LOCK#

The bridge is required to support **LOCK#** when a transaction is initiated on its primary bus (and is using the lock protocol), but is not required to support **LOCK#** on transactions that are initiated on its secondary bus. If a locked transaction is initiated on the primary bus and the bridge is the target, the bridge must adhere to the lock semantics defined by this specification. The bridge is required to complete (push) all PMWs (accepted from the primary bus) onto the secondary bus before attempting the lock on the secondary bus. The bridge may discard any requests enqueued (but not yet attempted on the secondary bus), allow the locked transaction to pass the enqueued requests, or simply complete all enqueued transactions before attempting the locked transaction on the secondary interface. Once a locked transaction has been enqueued by the bridge, the bridge cannot accept any other transaction from the primary interface until the lock has completed except for a continuation of the lock itself by the lock master. Until the lock is established on the secondary interface, the bridge is allowed to continue enqueuing transactions from the secondary interface, but not the primary interface. Once lock has been established on the secondary interface, the bridge cannot accept any posted write data moving toward the primary interface until **LOCK#** has been released (**FRAME#** and **LOCK#** deasserted on the same rising clock edge). (In the simplest implementation, the bridge does not accept any other transactions in either direction once lock is established on the secondary bus except for locked transactions from the lock master.) The bridge must complete previously enqueued PMW, DRC, and DWC transactions moving toward the primary bus before allowing the locked access to complete on the originating bus.

## Error Conditions

A bridge is free to discard data or status of a transaction that was completed using Delayed Transaction termination when the master has not repeated the request within $2^{10}$ PCI clocks (about 30 μs at 33 MHz). However, it is recommended that the bridge not discard the transaction until $2^{15}$ PCI clocks (about 983 μs at 33 MHz) after it acquired the data or status. The shorter number is useful in system where a master designed to a previous version of this specification frequently fails to repeat a transaction exactly as first requested. In this case, the bridge may be programmed to discard the abandoned Delayed Completion early and allow other transactions to proceed. Normally, however, the bridge would wait the longer time in case the repeat of the transaction is being delayed by another bridge or bridges designed to a previous version of this specification that did not support Delayed Transactions.

When this timer (referred to as the Discard Timer) expires, the device is required to discard the data; otherwise, a deadlock may occur.

> Note: When the transaction is discarded, data may be destroyed. This occurs when the discarded Delayed Completion is a read to a non-prefetchable region.

When the Discard Timer expires, the device may choose to report or ignore the error. When the data is prefetchable, it is recommended that the device ignore the error since system integrity is not affected. However, when the data is not prefetchable, it is recommended that the device report the error to its device driver since system integrity is affected. A bridge may assert **SERR#** since it does not have a device driver.

## Illustrations of the Use of the Ordering Rules

To illustrate the use of the ordering rules, consider the following examples. Each example shows a sequence of transactions on each bus, with time advances from left to right. Generally no attempt is made to align the time scale of the two buses:

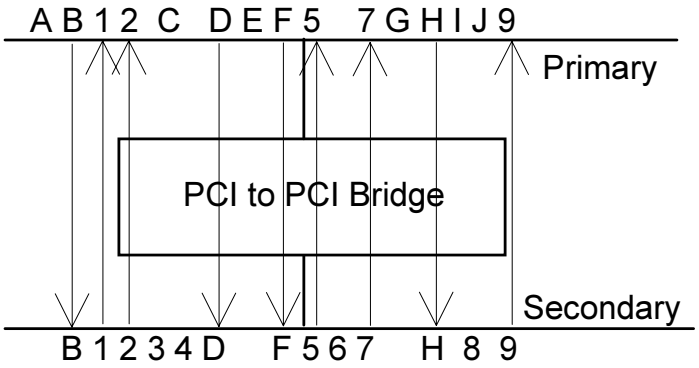



**Figure 5-2: Transaction Ordering Example 1**

In Figure 5-2, there are two streams of transactions, one on the primary bus and one on the secondary bus. Transactions that originate on the primary bus are designated by letters and complete in the sequence [A B C D E F G H I J]. Transactions that originate on the secondary bus are designated by numbers, and complete in the sequence [1 2 3 4 5 6 7 8 9]. Arrows are used to indicate that a transaction is forwarded from one bus to the other. The bridge forwards transaction B, D, F, and H from the primary bus to the secondary bus and transactions 1, 2, 5, 7, and 9 from the secondary bus to the primary bus. The resulting sequence on the primary bus is [A B 1 2 C D E F 5 7 G H I J 9] while the sequence on the secondary bus is [B 1 2 3 4 D F 5 6 7 H 8 9]. Notice that in this example, the arrows never cross, which indicates that in this example, the order of the transactions does not change independent of which bus the transaction initiates on or targets.

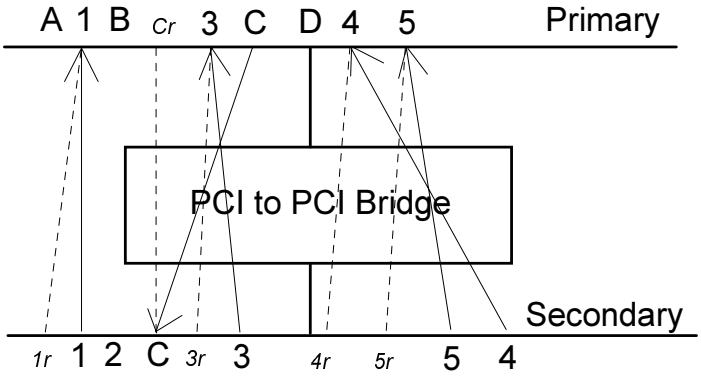



**Figure 5-3: Transaction Ordering Example 2**

To illustrate how the order of certain transactions can change when crossing a bridge, Figure 5-3 shows several Delayed Transactions. Initial enqueuing of the Delayed Request is designated in

italics with a small "*r*" and connected to the completed transaction on the destination bus with a dotted line. This illustrates when the master first requested the transactions but was terminated with Retry when the bridge enqueued the Delayed Request. As before, arrows indicate transactions that cross the bridge and point toward the destination bus. The head of the arrows shows when the transaction completed on the destination bus, and the tail of the arrow shows when the transaction completed on the originating bus. Delayed Transactions 3 and C are crossing the bridge in opposite directions. Although transactions C is terminated with Retry on the primary bus and completes on the secondary bus before transaction 3, it does not complete on the primary bus until after transaction 3. Similarly, Delayed Transactions crossing in the same direction can be reordered. Transactions 4 and 5 complete on the primary bus in the same order as they were terminated with Retry on the secondary bus, but they complete on the secondary bus in the opposite order. This reordering can occur because it depends upon such things as PCI bus arbitration sequence and timing of the master's request to repeat the transaction on the secondary bus relative to when it completed on the primary bus. In summary, the order in which Delayed Transactions start and complete with respect to each other on either bus is irrelevant.

The relevant transaction ordering for a bridge involves posted memory write transactions. As indicated in Table 5-2, the order relative to posted memory writes of most transactions moving in either direction must be maintained by the bridge.

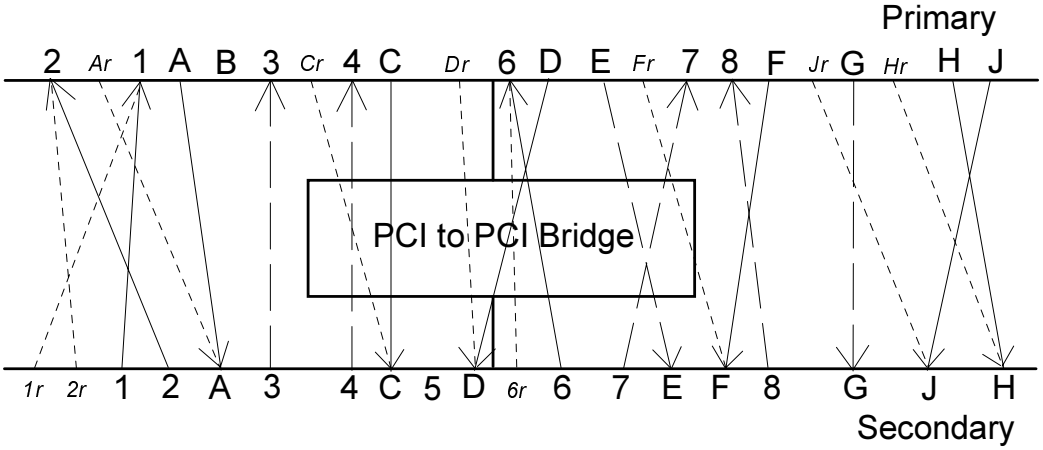



**Figure 5-4: Transaction Ordering Example 3**

Figure 5-4 illustrates some of the ordering cases involving posted memory writes. Transactions 3, 4, 7, 8, E, and G are posted memory writes, and are shown with long-dashed lines. The remaining transactions are all Delayed Transactions. Notice that posted memory write transactions crossing the bridge in the same directions complete on both busses in the same order, in this example [3 4 7 8] and [E G]. However, the order of a posted memory write with respect to another posted memory write crossing the bridge in the opposite direction is not important ([3 4 E 7 8 G] on primary side and [3 4 7 E 8 G] on secondary side).

No Delayed Request is allowed to pass a posted memory write going in the same direction, but posted memory writes are allowed to pass Delayed Requests. Delayed Request *Fr* is enqueued after posted memory write E and, therefore, Delayed Transaction F cannot complete on the secondary bus until after posted memory write E. However, posted memory write G can complete on the secondary bus before Delayed Transaction J, even though G completed on the primary bus after Delayed Request *Jr* was enqueued.

Figure 5-4 also illustrates that no Delayed Completion is allowed to pass a posted memory write going in the same direction, but posted memory writes are allowed to pass Delayed Completions. Since Delayed Transaction C completed after posted memory write 4 on the secondary bus, the Delayed Transaction cannot complete before the write on the primary bus. But even though posted memory write 8 completed on the secondary bus after Delayed Transaction F, the write can complete before the Delayed Transaction on the primary bus.

Besides maintaining a consistent view of write data, the ordering rules also avoid deadlocks. Unless a locked sequence is in progress, in general, the bridge must accept a posted memory write transaction addressing a target across the bridge, regardless of what other transactions preceded it on either side. The only exception is for temporary conditions like emptying the buffer of previous posted memory writes. The bridge may not continually terminate a memory write transaction with Retry while waiting for a non-locked transaction to complete in either direction.

## 5.6. Special Design Considerations

### 5.6.1. Read Starvation

Bridges designed to an earlier version of this specification do not implement Delayed Transactions, and typically do not meet the latency requirements of the *PCI Local Bus Specification* and can in normal operation starve masters on one side of the bus from fair access to the other side.

Consider, for example, the case where a single master on the secondary bus is executing a long string of write transactions addressing main memory, and during this time the CPU attempts to read from a target on the secondary bus. Since there are no other masters on the secondary bus, the writing master will quickly acquire the bus and post the first write transaction in the bridge. While the bridge is waiting to acquire the primary bus, the CPU is granted the bus and attempts to read from the secondary target. The ordering rules require the bridge to empty its posting buffer before allowing a read to complete, so it must Retry the CPU read. If the bridge does not implement Delayed Transactions, this CPU read is not enqueued and has made no forward progress through the bridge. When the bridge acquires the bus, it will execute its write and empty its posting buffer.

Since there are no other masters on the secondary bus in this example, it is quite possible for the same writing master to reacquire the secondary bus and refill the bridge's posting buffer before the CPU has a chance to repeat its read transaction. If this happens, then the sequence will repeat, starving the CPU until the master eventually finishes all of its write operations. Note that in some applications, such as a live video frame grabber, the write stream never stops.

This condition is avoided if the bridge executes the CPU read as a Delayed Transaction. In this case, when the CPU read is terminated with Retry, the bridge would enqueue a Delayed Read Request. If the secondary arbiter treats this request fairly with respect to secondary master requests for the bus, the CPU read would execute to completion on the secondary bus in between master writes to the bridge, even though the CPU had been terminated with Retry on the primary bus. The read data is then inserted between data of the write stream. The read data then reaches the primary bus where it is held until the CPU repeats the request. To complete the read

transaction the CPU need only reacquire the primary bus and repeat the transaction some time after the read data reaches the primary bus queue. Write data enqueued after the read data is accepted is allowed to pass the read completion if blocking occurs to avoid a deadlock.

## 5.6.2. Stale Data

It is the responsibility of the bridge to guarantee that any data provided to a master be current as of the time the master first attempted the read transaction. In general, this means that the bridge must discard the balance of any data prefetched on behalf of a master, but not taken when the master completed the transaction.

### Implementation Note: Stale-Data Problems Caused by Not Discarding Prefetch Data

Suppose a CPU has two buffers in adjacent main memory locations. The CPU prepares a message for a bus master in the first buffer and then signals the bus master to pick up the message. When the bus master reads its message, a bridge between the bus master and main memory prefetches subsequent addresses including the second buffer location.

Sometime later, the CPU prepares a second message using the second buffer in main memory and signals the bus master to come and get it. If the intervening bridge has not flushed the balance of the previous prefetch, then when the master attempts to read the second buffer, the bridge may deliver stale data.

Similarly, if a device were to poll a location behind a bridge and the bridge did not flush the buffer after each read by the device, the device would never observe a new value for the polled location

The Special Design Considerations section of the *PCI Local Bus Specification*, describes another situation in which a master might see stale data. If two masters are polling the same location using the same address, command, and byte enables, and one of the masters also writes to the location, the next read by the writing master may read the value before the write rather than after it. This same problem can occur if the two masters are not sharing the same location, but use the same address, command, and byte enables, because one of the masters starts reading at a smaller address than the one it actually wants.

Since the bridge, in general, has no knowledge of which master actually is making the request, the bridge has no alternative but to supply the read completion data to the first master to repeat the identical read transaction. Although it is difficult to envision a real application that would behave this way, if one exists, then it is that application designer's responsibility to avoid the problem by doing a dummy read of the location or device after writing to it.

## 5.6.3. Deadlocks

The PCI ordering rules permit memory write transactions to be posted anywhere in the system, and require that from time to time those posted writes must be flushed before other transactions are allowed to complete so that all masters in the system will have a consistent view of data. As a result, deadlocks can occur in numerous cases if targets do not follow the ordering rules for accepting posted memory writes. In almost all cases, it is required that a target (including a

bridge) accept a posted memory write addressed to it regardless of what other non-locked transactions may have preceded it. The only exception is for conditions that are guaranteed to be resolved over time; for example, while all buffers are filled with previous posted memory write transactions.

For example, suppose there are two bridges connected hierarchically, with Master A and Target 1 at the top, and Master B and Target 2 at the bottom as shown in Figure 5-5. Further suppose that Master A executes a memory write to Target 2, which is posted in the upper bridge, and Master B executes a read from Target 1, which crosses the lower bridge and appears at the secondary interface of the upper bridge. To satisfy the ordering rules listed above, bridge X must first empty the posted memory write addressed to Target 2 before it can complete the read from Target 1 on PCI Bus P. Bridge Z is required to accept this posted memory write even if the read transaction has already been attempted and terminated with Retry on bridge Z's upper bus (the PCI bus connecting bridges X and Z). This is true regardless of whether the read is executed as a Delayed Transaction or not. If the lower bridge were to require its read to complete before accepting the posted memory write, the system would deadlock. The *PCI Local Bus Specification*, Appendix E includes other examples of deadlocks which can occur if targets (including bridges) don't accept posted memory writes.

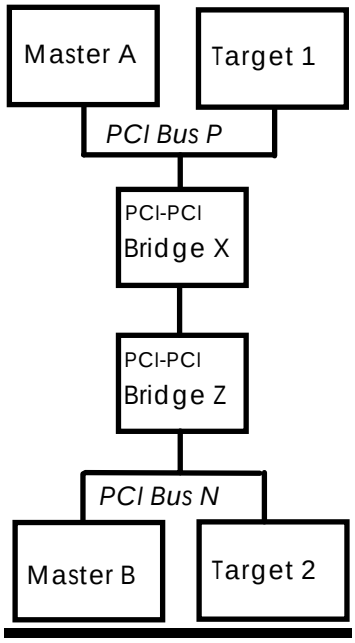



**Figure 5-5: Deadlock Example**

## 5.7. Combining Separate Writes Into a Single Burst Transaction

When a bridge forwards memory writes in either the memory mapped I/O or prefetchable memory address ranges, it is allowed to combine separate but sequential[5] memory writes into a single burst transfer (using linear increment addressing) provided the implied ordering is not changed. For example, separate writes to Dword 1, 2, and 4 can be combined and forwarded as a single burst (where the byte enables for Dword 3 are deasserted). However, separate writes to Dword 4, 3, and 1 cannot be combined into a burst but must be forwarded as three separate transactions in the same order as they were received.

Combining of I/O writes or configuration writes by a bridge is not allowed. Combining of memory writes by a bridge is optional. See the *PCI Local Bus Specification* for additional information on write combining.

## 5.8. Merging Separate Writes Into a Single Transaction

When a bridge forwards memory writes in a prefetchable memory address range, it is allowed to merge separate but sequential masked writes to one Dword address into a single data phase transfer, provided any byte location is written only once. For example, consider a sequence of separate byte writes to bytes 3, 1, 0, and 2 of a Dword. A bridge is allowed to merge these writes and forward them as a single write transaction. However, if the write sequence is byte 1, 1, 2, and 3, then the first write to byte 1 must be forwarded as a separate write transaction. The remaining byte writes could be forwarded as a single write with byte enables 1, 2, and 3 asserted and byte enable 0 deasserted.

Merging of I/O writes, configuration writes, or memory writes in the memory mapped I/O address range by a bridge is not allowed. Merging of memory writes in the prefetchable memory range by a bridge is optional. See the *PCI Local Bus Specification* for additional information on write merging.

## 5.9. Collapsing of Writes

When a bridge forwards write transactions, it cannot collapse sequential writes to the same address into a single transfer. Two sequential write transactions to the same address in which at least one byte enable has been asserted in both transactions must be forwarded as separate write transactions by a bridge.

Collapsing of writes of any type by a bridge is not allowed. See the *PCI Local Bus Specification* for additional information on write collapsing.

[5]The term sequential is used to indicate that the events (PCI transactions) occur in the order indicated without any other intervening transactions.

# Chapter 6
# Error Support

## 6.1. Introduction

There are many types of errors that a PCI-to-PCI bridge can detect or report. Possible errors include:

- Address parity errors
- Data parity errors
- Master-Aborts
- Target-Aborts
- Discard Timer timeout errors (when Delayed Transactions are supported)
- Secondary interface **SERR#** assertions (by other devices)

A bridge will respond as a target to a PCI transaction on its primary interface when its configuration registers, expansion ROM (if supported), or internal registers mapped by its Base Address Registers (if supported) are accessed. When the bridge itself is the target of a transaction (it is not forwarding a transaction), its error behavior must comply with the requirements of the *PCI Local Bus Specification*.

A bridge will also respond as a intermediate target when forwarding a transaction to its opposite PCI interface. Similarly, a bridge will operate as an intermediate master when it is forwarding a transaction from its opposite interface. The presence of a bridge in the path between the originating master and ultimate target of a PCI transaction must be transparent whenever possible. As a result, bridges attempt to propagate errors between the originating master and ultimate target so that the errors are seen by both. However, for some error conditions it is not possible for a bridge to be transparent. In these cases, the error handling requirements enable recovery at the lowest level possible as outlined in the *PCI Local Bus Specification*. Some errors force a bridge to exhibit non-transparent behavior in order to preclude more severe problems such as live-lock or deadlock conditions.

A PCI-to PCI bridge has two PCI interfaces and, therefore, additional sets of control and status bits are needed for the secondary interface. The following configuration register bits affect the error behavior of a bridge:

**Command Register (configuration register offset 04h)**

- Parity Error Response bit (bit 6) – enables parity error detection on the primary interface
- SERR# Enable bit (bit 8) – enables assertion of **SERR#** on the primary interface

**Bridge Control Register (configuration register offset 3Ch)**

- Parity Error Response bit (bit 0) – enables parity error detection on the secondary interface
- SERR# Enable bit (bit 1) – enables forwarding of **SERR#** from the secondary to the primary interface
- Discard Timer SERR# Enable bit (bit 11) – enables assertion of **SERR#** on the primary interface when a discard timer error occurs
- Primary Discard Timer bit (bit 8) – selects the number of PCI clocks that the bridge will wait for a master on the primary interface to repeat a Delayed Transaction request
- Secondary Discard Timer bit (bit 9) - selects the number of PCI clocks that the bridge will wait for a master on the secondary interface to repeat a Delayed Transaction request

A bridge uses the following configuration register bits to report error status:

**Status Register (configuration register offset 06h)**

- Master Data Parity Error bit (bit 8) – reports the detection of a data parity error while the bridge is the master of a transaction on the primary interface of the bridge
- Signaled System Error bit (bit 14) – reports the assertion of the primary interface **SERR#** by the bridge
- Detected Parity Error bit (bit 15) – reports the detection of a parity error on the primary interface of the bridge

**Secondary Status Register (configuration register offset 1Eh)**

- Master Data Parity Error bit (bit 8) – reports the detection of a data parity error while the bridge is a master of a transaction on the secondary interface of the bridge
- Received System Error bit (bit 14) – reports the detection of an **SERR#** assertion on the secondary interface of the bridge
- Detected Parity Error bit (bit 15) – reports the detection of a parity error on the secondary interface of the bridge

**Bridge Control Register (configuration register offset 3Ch)**

- Discard Timer Status bit (bit 10) - reports the discard of a Delayed Transaction from a queue in the bridge as a result of the expiration of either the Primary Discard Timer or the Secondary Discard Timer

## 6.2. Parity Errors

The *PCI Local Bus Specification* specifies the requirements for parity generation, detection, and reporting. Because a bridge has two interfaces, and forwards transactions from one interface to the other, additional requirements specific to a bridge are necessary. A bridge is required to implement parity generation and parity error detection on both interfaces. The parity error handling methods used by a bridge vary depending on the type of transaction during which the error occurs, the settings of the various error control bits (listed in Section 6.1.), and the transaction completion method used by the bridge (Delayed Transaction or immediate completion). The following sections describe the required behavior of a bridge for address and data parity errors. In some error cases, different behaviors are permitted, and the bridge designer is free to choose one of the specified options.

For correct parity error handling both the primary and secondary interfaces of the bridge must be configured consistently. If the Parity Error Response bit in the Command register (corresponds to the primary interface) is set, then the Parity Error Response bit in the Bridge Control register (corresponds to the secondary interface) must also be set. Conversely, if the Parity Error Response bit in the Command register is cleared, then the Parity Error Response bit in the Bridge Control register must also be cleared. Some errors will not be reported if the various **SERR#** enable bits are not set even if the Parity Error Response bits are both set.

### 6.2.1. Address Parity Errors

The bridge must detect address parity errors for all transactions on either interface. The response to address parity errors by a bridge is controlled by the Parity Error Response bits in the Command register and the Bridge Control register. During the address phase of a transaction, all PCI agents decode the address and command to determine if they should respond to the transaction by asserting **DEVSEL#**. When a target determines that it should respond to a transaction, is enabled to respond to parity errors, and detects an address parity error, the *PCI Local Bus Specification* allows the target to use one of the following transaction termination methods:

- claim the transaction (by asserting **DEVSEL#**) and terminate it as though the address was correct
- claim the transaction (by asserting **DEVSEL#**) and terminate it with a Target-Abort
- not claim the transaction (by not asserting **DEVSEL#**) and letting it terminate with a Master-Abort

Any of these transaction termination methods are allowed for a bridge when an address parity error occurs. A bridge is not allowed to terminate a transaction with Retry solely because an address parity error was detected.

When the bridge detects an address parity error on its *primary* interface it must:

- assert **SERR#** on the primary interface (if enabled to do so by the SERR# Enable bit and the Parity Error Response Enable bit in the Command register)
- set the Signaled SERR# bit in the Status register (if **SERR#** assertion was enabled)

- set the Detected Parity Error bit in the Status register (independent of the setting of the Parity Error Response Enable bit in the Command register)
- *if* the bridge claims the transaction by asserting **DEVSEL#** and terminates it by signaling a Target-Abort it must also set the Signaled Target-Abort bit in the Status register

When the bridge detects an address parity error on its *secondary* PCI interface it must:

- assert **SERR#** on the primary interface (if enabled to do so by the SERR# Enable bit in the Command register and the SERR# Enable and Parity Error Response Enable bits in the Bridge Control register)
- set the Signaled SERR# bit in the Status register (if **SERR#** assertion was enabled)
- set the Detected Parity Error bit in the Secondary Status register (independent of the setting of the Parity Error Response Enable bit in the Bridge Control register)
- *if* the bridge has claimed the cycle and terminates it by signaling a Target-Abort it must also set the Signaled Target-Abort bit in the Secondary Status register

## 6.2.2. Read Data Parity Errors

During a read transaction, the target device sources the data, and parity is not valid until **TRDY#** is asserted by the target device. As a result, a data parity error cannot occur during any attempt (initial or subsequent) by the master that is terminated with Retry. A data parity error can occur on the destination bus when read data is transferred from the target to the bridge (for subsequent delivery to the originating master). A data parity error can also occur on the originating bus when read data is transferred from the bridge to the originating master. The following sections describe the error handling methods used by a bridge for each of these cases. These error handling methods apply for all read transactions independent of the transaction completion method (Delayed Transaction or immediate completion).

## 6.2.2.1. Target Completion Error

This section describes the data parity error handling method used by a bridge when a data parity error occurs on the destination bus as read data is transferred from the target to the bridge (for subsequent delivery to the originating master). A data parity error that occurs during a particular data phase (or multiple data phases) of a burst read transaction on the destination bus must be forwarded by the bridge to the originating bus in the same data phase (or data phases) in which it occurred on the destination bus. The bridge must always forward the data and parity as read on the destination bus to the master on the originating bus to allow the originating master to observe the read data parity error condition. Note that the originating master will generally detect the parity error when the read data is transferred to the master by the bridge during the master completion phase (refer to Section 6.2.2.2.).

If a data parity error occurs on the *primary* interface when the bridge is acting as a bus master while forwarding a read transaction upstream, then the bridge must:

- assert **PERR#** on the primary bus (if enabled to do so by the Parity Error Response bit in the Command register)

- set the Master Data Parity Error bit in the Status register (if the Parity Error Response bit in the Command register is set)

If a data parity error occurs on the *secondary* interface when the bridge is acting as a bus master while forwarding a read transaction downstream, then the bridge must:

- assert **PERR#** on the secondary bus (if enabled to do so by the Parity Error Response bit in the Bridge Control register)
- set the Master Data Parity Error bit in the Secondary Status register (if the Parity Error Response bit in the Bridge Control register is set)

The requirements to assert **PERR#** on the destination bus and to set the Master Data Parity Error bit in the corresponding status register also apply to read data phases that are prefetched by the bridge even if the originating master does not consume the read data on the originating bus. However, the detection and reporting of a data parity error on the destination bus is independent from the detection and reporting of the data parity error on the originating bus. If the originating master resides on the *primary* interface, the error status bits in the Status register of the bridge are not affected by errors that occurred during read data phases that were prefetched on the secondary interface but were not subsequently consumed on the primary interface. If the originating master resides on the *secondary* interface, the error status bits in the Secondary Status register of the bridge are not affected by errors that occurred during read data phases that were prefetched on the primary interface but were not subsequently consumed on the secondary interface.

### 6.2.2.2. Master Completion Error

This section describes the data parity error handling method used by a bridge if an error occurs on the originating bus when the read data is transferred from the bridge to the originating master. A read data parity error can occur on the originating bus during transactions that access registers internal to the bridge or during transactions that are forwarded from one PCI interface of the bridge to the other. When the bridge forwards a transaction, data may transfer on the destination bus with or without an error. If the read data is transferred on the destination bus without a data parity error, an error may still be introduced on the originating bus.

If a data parity error is detected by the originating master when data is read from the bridge (either from an internal bridge register or from a buffer containing read data obtained from a read transaction that was forwarded across the bridge), the master must assert **PERR#** if enabled to do so by its Parity Error Response bit. The bridge takes no action and sets no status bits for such an error.

## 6.2.3. Non-Posted Write Data Parity Errors

During a write transaction, the master sources the write data and must assert **IRDY#** when the data is valid independent of the response by the target. Therefore, a data parity error can occur during any attempt (initial or subsequent) by the originating master that is terminated with Retry. A data parity error can also occur when a non-posted write transaction is completed on the destination bus by the bridge. In addition, it is possible for a data parity error to be constant (i.e., the same error occurs each time the master repeats the transaction) or transient (i.e., the error

occurs on some but not other repetitions of the transaction by the master). The error handling methods for a bridge are designed to detect and report both constant and transient data parity errors during non-posted writes and to prevent transient data parity errors from causing a live-lock or deadlock. The following sections describe the error handling methods used by a bridge for each of these cases.

### 6.2.3.1. Master Request Error

If the bridge detects a data parity error when responding as a target on its *primary* interface to a write transaction that would otherwise have been handled as a Delayed Transaction, the bridge must do the following:

- Complete the data phase in which the error occurred by asserting **TRDY#**. If the master is attempting a burst, the bridge must also assert **STOP#**.
- Report the error to the master by asserting **PERR#** (if enabled to do so by the Parity Error Response bit in the Command register).
- Set the Detected Parity Error bit in the Status register.
- Discard the transaction. No Delayed Write Request is enqueued and no Delayed Write Completion is retired.

If the bridge detects a data parity error when responding as a target on its *secondary* interface to a write transaction that would otherwise have been handled as a Delayed Transaction, the bridge must do the following:

- Complete the data phase in which the error occurred by asserting **TRDY#**. If the master is attempting a burst, the bridge must also assert **STOP#**.
- Report the error to the master by asserting **PERR#** (if enabled to do so by the Parity Error Response bit in the Bridge Control register).
- Set the Detected Parity Error bit in the Secondary Status register.
- Discard the transaction. No Delayed Write Request is enqueued and no Delayed Write Completion is retired.

When parity error response is enabled, the bridge will only enqueue a Delayed Write Request when a data parity error is ***not*** detected during the master request.

### 6.2.3.2. Target Completion Error

The bridge forwards a Delayed Write Request to the destination bus only if no error occurred during the master request. However, a data parity error may occur when the transaction is completed on the destination bus. In this case, the error is forwarded back to the originating master during the Master Completion phase of the transaction.

When a Delayed Write Request is forwarded upstream by the bridge to its *primary* interface, the target of the transaction asserts **PERR#** if it detects a data parity error (and is enabled to report parity errors). If **PERR#** is asserted by the target, then the bridge must:

- set the Master Data Parity Error bit in the Status register
- record the occurrence of the error in the corresponding Delayed Write Completion (produced by completion of the Delayed Write Request)

When the Delayed Write Completion is returned to the originating master on the *secondary* interface of the bridge, the bridge must reflect the occurrence of the error from the destination bus to the master by asserting **PERR#** (if enabled). The Detected Parity Error bit in the Secondary Status register *is not* set.

When a Delayed Write Request is forwarded downstream by the bridge to its *secondary* interface, the target of the transaction asserts **PERR#** if it detects a data parity error (and is enabled to report parity errors). If **PERR#** is asserted by the target, then the bridge must:

- set the Master Data Parity Error bit in the Secondary Status register
- record the occurrence of the error in the corresponding Delayed Write Completion (produced by completion of the Delayed Write Request)

When the Delayed Write Completion is returned to the originating master on the primary interface of the bridge, the bridge must reflect the occurrence of the error from the destination bus to the master by asserting **PERR#** (if enabled). The Detected Parity Error bit in the Status register *is not* set.

### 6.2.3.3. Master Completion Error

If the bridge enqueues a Delayed Write Request and later detects a data parity error during a subsequent repetition of the transaction by the originating master, the bridge terminates the repeated transaction as if it were a new transaction as described in Section 6.2.3.1. The bridge does not retire any Delayed Write Completions even if the transaction appears to match one previously enqueued (it is impossible to determine whether the transaction really matches a previously enqueued one since an error is present).

If a constant data parity error is present on all subsequent reattempts of a previously enqueued Delayed Write Request, then the bridge will eventually have an orphan Delayed Write Completion (as a result of the initial Delayed Request). The orphan completion is discarded when the discard timer expires (refer to Sections 5.3. and 6.5.). While waiting for the discard timer to expire, the bridge may not be able to accept a new Delayed Transaction since it is not required to handle multiple Delayed Transactions at the same time. However, since this condition is temporary, a deadlock cannot occur. While in this condition, the bridge is required to complete transactions that use Memory Write and Memory Write and Invalidate transactions (refer to Sections 5.5. and 5.6.3.).

## 6.2.4. Posted Write Data Parity Errors

There are two data parity error cases to consider when a bridge forwards a posted write transaction (a Memory Write or a Memory Write and Invalidate transaction). In the first case, the error occurs on the bus on which the posted write transaction originates. In the second case, the posted write transaction completes without error on the originating bus, but an error occurs when the transaction is completed on the destination bus. The following sections discuss the error handling methods used by a bridge for each of these cases.

### 6.2.4.1. Originating Bus Error

This section describes the data parity error handling method used by a bridge if an error occurs when write data is transferred from the originating master to the bridge during a posted write transaction. This data parity error can occur during transactions that access registers internal to the bridge or during transactions that are forwarded from one PCI interface of the bridge to the other. If the transaction is forwarded by the bridge, the bridge must always forward the data and parity as received on the originating bus to the destination bus to allow the target to observe the write data parity error condition.

If a posted write transaction accesses registers internal to the bridge (i.e., it is not forwarded to the other interface by the bridge), then the bridge must adhere to the requirements of the *PCI Local Bus Specification* for reporting data parity errors that occur during the transaction. If the bridge is responding as the target on the *primary* interface and it detects a write data parity error, it must assert primary **PERR#** (if enabled to do so) and set the Detected Parity Error bit in the Status register. If the bridge is responding as the target on the *secondary* interface and it detects a write data parity error, it must assert secondary **PERR#** (if enabled to do so) and set the Detected Parity Error bit in the Secondary Status register.

If the bridge detects a data parity error when responding as a target to a downstream posted write transaction on the *primary* bus, it must:

- assert **PERR#** on the primary bus
- retain the bad parity and data in its write buffers
- set the Detected Parity Error bit in the Status register

When the posted write transaction is initiated on the secondary bus by the bridge, the target of the transaction asserts **PERR#** (if enabled) provided it also detects the error. If **PERR#** is asserted by the target, then the bridge must set the Master Data Parity Error bit in the Secondary Status register.

If the bridge detects a data parity error when responding as a target to an upstream posted write transaction on the *secondary* bus, it must:

- assert **PERR#** on the secondary bus
- retain the bad parity and data in its write buffers
- set the Detected Parity Error bit in the Secondary Status register

When the posted write transaction is initiated on the primary bus, the target of the transaction asserts **PERR#** (if enabled) provided it also detects the error. If **PERR#** is asserted by the target, the bridge must set the Master Data Parity Error bit in the Status register.

### 6.2.4.2. Destination Bus Error

This section describes the data parity error handling method used by a bridge if an error occurs when write data is transferred from the bridge to the target during a posted write transaction forwarded by a bridge. If a data parity error is detected by the target during a write transaction, it asserts **PERR#** if enabled to do so by its Parity Error Response bit. In this case, if the target resides on the *primary* interface of the bridge and the Parity Error Response bit in the Command register of the bridge is set, the bridge must set the Master Data Parity Error bit in its Status register. Similarly, if the target resides on the *secondary* interface of the bridge, the bridge must set the Master Data Parity Error bit in the Secondary Status register.

Additional error response requirements apply when a posted write transaction completes without data parity errors on the originating bus (the bridge responds as the target), but data parity errors are reported when the transaction completes on the destination bus (the bridge is the master). In this case, the bridge *cannot* pass information on the error back to the originating master. Therefore, the bridge must also assert **SERR#** on the primary interface and set the Signaled System Error bit in the Status register when the following conditions are true:

- a data parity error *is not* detected by the bridge when it responds as the target on the originating bus.
- when the bridge completes the transaction (as the master) on the destination bus, the target detects and reports a data parity error by asserting **PERR#**.
- the **SERR#** Enable bit in the Command register is set.

## 6.3. Master-Aborts

A bridge provides two methods for handling a Master-Abort termination when it is the master of a transaction as controlled by the Master-Abort Mode bit in the Bridge Control register.

In the default case, the Master-Abort Mode bit is cleared and a Master-Abort is not considered to be an error unless it occurs during an exclusive access transaction (refer to Section 6.3.3.). However, when the Master-Abort mode bit is set, it is considered an error condition when the bridge is the master of any transaction type other than a Special Cycle and the transaction terminates with a Master-Abort. Master-Aborts are never reported when they occur during Special Cycle transactions.

### 6.3.1. Non-posted Transactions

When the Master-Abort Mode bit is cleared, the bridge will operate in a PC compatibility mode. When a non-exclusive read transaction crosses a bridge and is terminated on the destination bus by a Master-Abort, the bridge will return FFFF FFFFh to the originating master and terminate the read transaction on the originating bus normally (by asserting **TRDY#**). When a non-posted,

non-exclusive write transaction crosses a bridge and is terminated by a Master-Abort, the bridge will complete the write transaction on the originating bus normally (by asserting **TRDY#**) and discard the write data. In this case, if the bridge is forwarding the transaction upstream and it terminates with a Master-Abort on the *primary* interface, the bridge must set the Received Master-Abort bit in the Status register. Similarly, if the bridge is forwarding the transaction downstream and it terminates with a Master-Abort on the *secondary* interface, the bridge must set the Received Master-Abort bit in the Secondary Status register.

When the Master-Abort Mode bit is set, the bridge must signal a Target-Abort to the originating master of a read or a non-posted, non-exclusive write transaction when the corresponding transaction on the destination bus is terminated by a Master-Abort.

In this case, if the bridge is forwarding the transaction upstream and is terminated with a Master-Abort on the *primary* interface the bridge must:

- set the Received Master-Abort bit in the Status register
- terminate the corresponding transaction on the secondary bus by signaling a Target-Abort
- set the Signaled Target-Abort bit in the Secondary Status register

Similarly, if the bridge is forwarding the transaction downstream and is terminated with a Master-Abort on the *secondary* interface the bridge, must:

- set the Received Master-Abort bit in the Secondary Status register
- terminate the corresponding transaction on the primary bus by signaling a Target-Abort
- set the Signaled Target-Abort bit in the Status register

## 6.3.2. Posted Write Transactions

If the bridge is forwarding the transaction upstream and it terminates with a Master-Abort on the *primary* interface, the bridge must set the received Master-Abort bit in the Status register. If the bridge is forwarding the transaction downstream and it terminates with a Master-Abort on the *secondary* interface, the bridge must set the received Master-Abort bit in the Secondary Status register. When a Master-Abort is detected on the destination bus by the bridge when forwarding a posted write burst transaction and the transaction is still in progress on the originating bus, it is recommended that the bridge terminate the transaction on the originating bus as soon as possible. When a posted write transaction forwarded by the bridge terminates in a Master-Abort, the bridge must discard any remaining write data for that transaction (the current data phase and any additional data phases if a burst transaction.)

When the Master-Abort Mode bit is cleared and a posted write transaction forwarded by the bridge terminates with a Master-Abort, no error is reported (note that the Master-Abort bit is still set). When the Master-Abort Mode bit is set and a posted write transaction forwarded by the bridge terminates with a Master-Abort on the destination bus, the bridge must:

- assert **SERR#** on the primary interface
- set the Signaled System Error bit in the Status register (if enabled by the SERR# Enable bit in the Command register)

### 6.3.3. Exclusive Access Master-Abort

A bridge must terminate a non-posted exclusive access transaction on its primary interface with Target-Abort when the transaction terminates with Master-Abort when forwarded to the secondary interface of the bridge (independent of the Master-Abort Mode bit in the Bridge Control register). This requirement is necessary to avoid potential deadlock conditions. Note that exclusive access transactions are supported as downstream transactions only.

## 6.4. Target-Aborts

A bridge may signal a Target-Abort under certain error conditions when it is responding as the target of a transaction. These include:

- error conditions that occur internal to the bridge when the bridge is responding as the target
- Target-Aborts detected on the destination bus by the bridge when completing a non-posted transaction
- Master-Aborts that occur on the destination bus when the bridge is completing an exclusive read transaction

These conditions are discussed in the follow sections and in Section 6.3.3..

### 6.4.1. Internal Errors

When a fatal error occurs internal to a bridge that prevents the bridge from completing a transaction as a target, it must terminate the transaction by signaling a Target-Abort. If the error occurs on primary interface of the bridge, it must set the Signaled Target-Abort bit in the Status register. If the error occurs on the secondary interface of the bridge, it must set the Signaled Target-Abort bit in the Secondary Status register.

### 6.4.2. Non-Posted Write Transactions

When forwarding any transaction except for a posted write transaction, a Target-Abort condition detected by the bridge on the destination bus must be returned to the originating master on the originating bus as a Target-Abort. If the error occurs during a burst transaction, the error must be signaled on the originating bus in the same data phase in which it occurred on the destination bus.

If a Target-Abort occurs on the *primary* interface when the bridge is acting as a bus master while forwarding a non-posted write transaction upstream, the bridge must:

- set the Received Target-Abort bit in the Status register
- complete the corresponding data phase on the secondary interface by signaling a Target-Abort
- set the Signaled Target-Abort bit in the Secondary Status register

If a Target-Abort occurs on the *secondary* interface when the bridge is acting as a bus master while forwarding a non-posted write transaction downstream, the bridge must:

- set the Received Target-Abort bit in the Secondary Status register
- complete the corresponding data phase on the primary interface by signaling a Target-Abort
- set the Signaled Target-Abort bit in the Status register

### 6.4.3. Posted Write Transactions

When a bridge forwards a posted write transaction, the target termination for any data phase cannot be returned to the originating master directly for one of the following reasons:

- The posted write transaction on the originating bus may have already terminated.
- During a posted write transaction, data is transferred on the originating bus first and at a later time on the destination bus. Thus, it is not possible for the bridge to report the target completion status for a particular data phase on the destination bus in the corresponding data phase on the originating bus.

If a Target-Abort occurs on the primary interface when the bridge is acting as a bus master while forwarding a posted write transaction upstream, the bridge must:

- set the Received Target-Abort bit in the Status register
- *if* the SERR# Enable bit in the Command register is set the bridge must also:
  - assert **SERR#** on the primary interface
  - set the System Error Signaled bit in the Status register

If a Target-Abort occurs on the secondary interface when the bridge is acting as a bus master while forwarding a posted write transaction downstream, the bridge must:

- set the Received Target-Abort bit in the Secondary Status register
- *if* the SERR# Enable bit in the Command Register is set the bridge must also:
  - assert **SERR#** on the primary interface
  - set the System Error Signaled bit in the Status Register

## 6.5. Discard Timer Timeout Errors

When a Delayed Transaction completes on the destination bus, it becomes a Delayed Completion. Once all ordering requirements have been satisfied and the bridge is ready to complete the Delayed Transaction with the originating master, a timer referred to as the discard timer is enabled to count. To avoid a potential deadlock, the bridge is required to delete the Delayed Completion from its queue if the originating master does not repeat the transaction before the discard timer expires. A bridge has discard timers for both its primary and secondary interfaces, the Primary Discard Timer and the Secondary Discard Timer, respectively.

The Primary Discard Timer is used to monitor the time it takes for a master on the primary interface of the bridge to reattempt the transaction corresponding to the Delayed Completion. The duration of the interval timed by the Primary Discard Timer is controlled by the Primary Discard Timer bit in the Bridge Control register.

The Secondary Discard Timer is used to monitor the time it takes for a master on the secondary interface of the bridge to reattempt the transaction corresponding to the Delayed Completion. The duration of the interval timed by the Secondary Discard Timer is controlled by the Secondary Discard Timer bit in the Bridge Control register.

When either the Primary Discard Timer or Secondary Discard Timer expires, the Discard Timer Status bit in the Bridge Control register must be set. When the Discard Timer SERR# Enable bit in the Bridge Control register is set and the SERR# Enable bit in the Command register is set, the bridge must also assert **SERR#** on the primary interface of the bridge and set the Signaled System Error bit in the Status register.

## 6.6. Secondary Interface SERR# Assertions

A bridge never asserts **SERR#** on its secondary interface (secondary **SERR#** is an input only signal). Whenever a bridge detects the assertion of **SERR#** on its secondary interface (by another agent), the bridge must set the Received System Error bit in the Secondary Status register. A bridge is enabled to propagate the secondary **SERR#** assertion upstream by asserting **SERR#** on the primary interface when all the conditions below are met.

- **SERR#** is asserted on the secondary bus;
- the Secondary SERR# Enable bit in the Bridge Control register is set (**SERR#** is enabled to be forwarded); and
- the SERR# Enable bit in the Command register is set (**SERR#** is enabled to be asserted on the primary interface).

This allows **SERR#** assertions to be propagated upstream through a hierarchy of bridges.

Whenever a bridge asserts **SERR#** on its primary interface, it must set the Signaled System Error bit in the Status register. If either SERR# Enable bit (Bridge Control register or Command register) is cleared, then **SERR#** assertions on the secondary interface do not result in **SERR#** assertions on the primary interface and do not affect the Signaled System Error bit in the Status register.

# Chapter 7
# PCI Bus Commands

## 7.1. Summary of Bridge Transaction Command Support

Table 7-1 summarizes the support defined or allowed by this specification for a bridge for each PCI bus command for the one of following four operating cases:

- the bridge initiating a transaction with the bus command as a *master* on the *primary* interface
- the bridge responding to a transaction with the bus command as a *target* on the *primary* interface
- the bridge initiating a transaction with the bus command as a *master* on the *secondary* interface
- the bridge responding to a transaction with the bus command as a *target* on the *secondary* interface

The symbols used in the table entries are defined below:

- NA – Not allowed. This specification does not allow any use of the bus command.
- REQ – Required. This specification requires support of the bus command by a bridge for the operating case.
- OPT – Optional. This specification defines optional support of the bus command by bridge for the operating case. An implementation may optionally support the bus command for the operating case but must do so as specified by this document.

**Table 7-1: Commands Supported By Bridge Interface**

| C/BE[3::0]# | | Primary Interface | | Secondary Interface | |
|---|---|---|---|---|---|
| | | Master | Target | Master | Target |
| 0000 | Interrupt Acknowledge | NA | NA | NA | NA |
| 0001 | Special Cycle | REQ[1] | NA | REQ[2] | NA |
| 0010 | I/O Read | OPT | OPT | OPT | OPT |
| 0011 | I/O Write | OPT | OPT | OPT | OPT |
| 0100 | Reserved | NA | NA | NA | NA |
| 0101 | Reserved | NA | NA | NA | NA |
| 0110 | Memory Read | REQ | REQ | REQ | REQ |
| 0111 | Memory Write | REQ | REQ | REQ | REQ |
| 1000 | Reserved | NA | NA | NA | NA |
| 1001 | Reserved | NA | NA | NA | NA |
| 1010 | Configuration Read | NA | REQ | REQ | NA |
| 1011 | Configuration Write | REQ[3] | REQ | REQ | REQ[4] |
| 1100 | Memory Read Multiple | OPT | OPT[5] | OPT | OPT[5] |
| 1101 | Dual Address Cycle | OPT | OPT | OPT | OPT |
| 1110 | Memory Read Line | OPT | OPT[6] | OPT | OPT[6] |
| 1111 | Memory Write and Invalidate | OPT | OPT[7] | OPT | OPT[7] |

Notes:

1. Only initiated by the bridge when forwarding a Type 1 Configuration transaction that specifies a conversion to a Special Cycle on the primary bus.
2. Only initiated by the bridge when forwarding a Type 1 Configuration transaction that specifies conversion to a Special Cycle transaction on the secondary bus.
3. Only initiated by the bridge when forwarding a Type 1 Configuration transaction that specifies conversion to a Special Cycle transaction with a destination other than the primary bus.
4. Only responded to by the bridge when forwarding a Type 1 Configuration transaction that specifies conversion to a Special Cycle Transaction on an upstream bus segment.
5. If the Memory Read Multiple command is not supported by the bridge, the bridge must alias the command to a Memory Read or a Memory Read Line command when responding as a target.
6. If the Memory Read Line command is not supported by the bridge, the bridge must alias the command to a Memory Read or a Memory Read Multiple command when responding as a target.
7. If the Memory Write and Invalidate command is not supported by the bridge, the bridge must alias the command to a Memory Write command when responding as a target.

# Chapter 8
# Arbitration and Latency Requirements

## 8.1. Bridge Interface Priority

Because of the initial latency requirements of the *PCI Local Bus Specification*, bridges are required to implement[6] Delayed Transactions to complete non-posted transactions that cross from one interface to the other. Bridges are not required to give one interface priority over the other but are required to allow fair arbitration between the interfaces.

## 8.2. Secondary Interface Arbitration Requirements

Every bus segment in a PCI system requires an arbiter. Since a bridge creates a new bus segment, it is anticipated that an arbiter will be a common bridge feature. However, a bridge is not required to provide an integral arbiter.

It is recommended that a bridge provide the arbiter for the secondary bus. If the bridge does provide the arbiter, it must adhere to the arbitration requirements of the *PCI Local Bus Specification*. The arbiter is required to use some type of fairness algorithm. The following excerpt is from the *PCI Local Bus Specification* and is included here for convenience of the reader. Refer to the *PCI Local Bus Specification* for full details.

"An agent requests the bus by asserting its **REQ#**. Agents must only use **REQ#** to signal a true need to use the bus. An agent must never use **REQ#** to "park" itself on the bus. If bus parking is implemented, it is the arbiter that designates the default owner. When the arbiter determines an agent may use the bus, it asserts the agent's **GNT#**.

The arbiter may deassert an agent's **GNT#** on any clock. An agent must ensure its **GNT#** is asserted on the rising clock edge it wants to start a transaction. Note: A master is allowed to start a transaction when its **GNT#** is asserted and the bus is in an Idle state independent of the state of

[6] There are a few conditions where a bridge can meet the initial latency requirements for non-posted transactions without using Delayed Transactions. If this option is used, the bridge must give priority to downstream accesses when requests reach the bridge on both interfaces at the same time.

its **REQ#**. If **GNT#** is deasserted, the transaction must not proceed. Once asserted, **GNT#** may be deasserted according to the following rules.

1. If **GNT#** is deasserted and **FRAME#** is asserted, the bus transaction is valid and will continue.
2. One **GNT#** can be deasserted coincident with another **GNT#** being asserted if the bus is not in the Idle state. Otherwise, a one clock delay is required between the deassertion of a **GNT#** and the assertion of the next **GNT#**, or else there may be contention on the **AD** lines and **PAR** due to the current master doing address stepping.
3. While **FRAME#** is deasserted, **GNT#** may be deasserted at any time in order to service a higher priority[7] master or in response to the associated **REQ#** being deasserted.

## 8.3. Bus Parking

The arbiter is required to park the bus at a master to keep the **AD** and **C/BE#** buses and **PAR** from floating when the bus is Idle for long periods. The arbiter may park the bus at any agent (master) that is present on the secondary bus; however, it is recommended that the arbiter park the bus at the bridge for the following reasons:

- By parking the bus at the bridge, transactions that originate on the primary bus can complete faster on the secondary bus. The motivation for this is that the bandwidth on the primary bus is typically more important than bandwidth on the secondary bus.
- When a bridge is used to support more connectors (expansion boards), the arbiter may not know if a master resides on the card or if a card is actually present (without special logic).

## 8.4. Latency Requirements

A bridge must adhere to the latency requirements of the *PCI Local Bus Specification*. These requirements include the following:

- Target Initial Latency
- Target Subsequent Latency
- Master Data Latency
- Memory Write Maximum Completion Time (for internal bridge registers)
- Master Latency Timer Timeout

When the bridge is responding as a target to a transaction, it must complete the initial data phase within 16 clocks (Target Initial Latency) and terminate subsequent data phases (of the same transaction) within eight clocks (Target Subsequent Latency). To comply with these target latency requirements, a bridge must use Delayed Transactions to complete non-posted transactions that cross the bridge.

[7] Higher priority here does not imply a fixed priority arbitration, but refers to the agent that would win arbitration at a given instant in time.

Master Data Latency is the number of clocks the master takes to assert **IRDY#** indicating it is ready to complete the data phase and transfer data. All PCI devices including bridges are required to assert **IRDY#** within eight clocks of the assertion of **FRAME#** on the initial data phase and within eight clocks on all subsequent data phases.

The *PCI Local Bus Specification* requires targets to complete at least one data phase of a Memory Write or Memory Write and Invalidate transaction within a specified number of PCI clocks (Maximum Completion Time) but grants an exception to bridges when the transaction crosses a bridge. However, a bridge must adhere to the Maximum Completion Time (334 clocks at 33 MHz or slower and 668 clocks at 66 MHz) when a Memory Write or Memory Write and Invalidate transaction accesses a location within (or associated with) the bridge. The Maximum Completion Time requirement is not in effect during device initialization time, which is defined as the $2^{25}$ PCI clocks immediately following the deassertion of **RST#**.

The Master Latency Timer limits the tenure of a PCI bus master when it is using the bus. When the bridge is a master on the primary interface, it must relinquish the bus when its primary interface **GNT#** has been deasserted and the Master Latency Timer expires. When the bridge is a master on the secondary interface, it must relinquish the bus when its secondary interface **GNT#**[8] has been deasserted and the Secondary Master Latency Timer expires. When the Master Latency Timer expires and **GNT#** is deasserted while the bridge is the master of a Memory Write and Invalidate transaction, it must ignore the timeout condition until a cacheline boundary is reached (see the *PCI Local Bus Specification* for more details).

[8] When the bridge provides an arbiter for the secondary bus, this grant may be an internal signal.

# Chapter 9
# Interrupt Support

## 9.1. Interrupt Routing

A bridge is not required to route interrupts that originate on the PCI bus connected to the secondary interface of the bridge through the bridge. The *PCI Local Bus Specification* requires the interrupt handler (service routine), or the device which originates the interrupt, to guarantee that all buffers are flushed between the device and the final destination. This can be accomplished by the interrupt service routine of the device driver by performing a read of the device or by the device itself performing a read of the location last written by the device. In either case, the read will force buffers between the device and the final destination to be flushed. No special buffer flushing requirements exist for devices that use Message Signaled Interrupts, as defined by the *PCI Local Bus Specification*. Interrupt messages naturally flush buffers.

However, since bridges will be used on expansion boards, the BIOS will assume an association between device location and which **INTx#** line it uses when requesting an interrupt. Since only the BIOS knows how PCI **INTx#** lines are routed to the system interrupt controller, a mechanism is required to inform the device driver which IRQ its device will request an interrupt on. The Interrupt Line register (see Section 3.2.5.15.) is used to store this information. The BIOS code will assume the following binding behind the bridge and will write the IRQ number in each device as described in Table 9-1. The interrupt binding defined in this table is mandatory for expansion boards utilizing a bridge.

**Table 9-1: Interrupt Binding for Devices Behind a Bridge**

| Device Number on Add-in Bus | Interrupt Pin on Device | Interrupt Pin on Connector |
|---|---|---|
| 0, 4, 8, 12, 16, 20, 24, 28 | INTA# | INTA# |
| | INTB# | INTB# |
| | INTC# | INTC# |
| | INTD# | INTD# |
| 1, 5, 9, 13, 17, 21, 25, 29 | INTA# | INTB# |
| | INTB# | INTC# |
| | INTC# | INTD# |
| | INTD# | INTA# |
| 2, 6, 10, 14, 18, 22, 26, 30 | INTA# | INTC# |
| | INTB# | INTD# |
| | INTC# | INTA# |
| | INTD# | INTB# |
| 3, 7, 11, 15, 19, 23, 27, 31 | INTA# | INTD# |
| | INTB# | INTA# |
| | INTC# | INTB# |
| | INTD# | INTC# |

Device 0 on a secondary bus will have its **INTA#** line connected to the **INTA#** line of the connector. Device 1 will have its **INTA#** line connected to **INTB#** of the connector. This sequence continues and then wraps around once **INTD#** has been assigned.

When POST code is initializing the system, it assumes the previous routing information for devices on an expansion board that utilizes a bridge. POST code writes the appropriate IRQ information in each device's Interrupt Line register.

Note that Table 9-1 does not specify the routing of a bridge's interrupt pin (if implemented) to the interrupt pins of the add-in connector. Assuming that the bridge is a single function device, its interrupt pin is required to be connected to **INTA#** by the *PCI Local Bus Specification*.

# Chapter 10
# Signal Pins

## 10.1. Primary PCI Interface

### 10.1.1. Required Signals

The primary bus requires 50 signals to support PCI. These signals are listed below according to signal type.

Signal type:

| | |
|---|---|
| s/t/s | **FRAME#**, **IRDY#**, **TRDY#**, **STOP#**, **DEVSEL#**, **PERR#** |
| t/s | **AD[31::00]**, **PAR**, **C/BE[3::0]#**, **REQ#** |
| Input | **IDSEL**, **CLK**, **RST#**, **GNT#**, **LOCK#**[9] |
| o/d | **SERR#** |

### 10.1.2. Optional Signals

The primary interface of the bridge may optionally support the 64-bit extensions, Power Management, JTAG, and 66 MHz operation to provide more functionality or performance as needed. The bridge may also provide an interrupt pin, if necessary, to support implementation-specific functions.

Signal type:

| | |
|---|---|
| s/t/s | **REQ64#**, **ACK64#** |
| t/s | **AD[63::32]**, **PAR64**, **C/BE[7::4]#** |
| Input | **TDI**, **TCK**, **TMS**, **TRST#**, **M66EN**[10] |

[9] Note that in the prior version of the *PCI-to-PCI Bridge Architecture Specification* **LOCK#** was defined as a s/t/s on the primary interface. However, the *PCI Local Bus Specification* constrains the use of **LOCK#** to downstream transactions.

| | |
|---|---|
| Output | **TDO** |
| o/d | **INTA#, PME#** |
| other | **3.3 Vaux** |

## 10.2. Secondary PCI Interface

### 10.2.1. Buffered Clocks

Bridges that buffer the primary interface **CLK** and provide buffered clocks for secondary bus devices are common. However, the bridge cannot be a perfect buffer and will introduce clock duty cycle skew on the buffered clocks. As a result, it is possible that the buffered output clocks of the bridge will not meet the clock duty cycle requirements of the *PCI Local Bus Specification*.

An expansion board that uses devices behind a bridge must accommodate the clock buffering requirements of that bridge. For example, if the bridge's clock buffer affects the duty cycle of **CLK**, the rest of the devices on the expansion board must accept the different duty cycle.

The system must always guarantee the timing parameters for **CLK** at the input of the device in a PCI expansion slot, even if the motherboard places PCI expansion slots on the secondary side of a bridge. It is the responsibility of the motherboard designer to choose clock sources and bridges that will guarantee the timing parameters for **CLK** for all slots.

Below are recommendations to designers to minimize the effects of clock skew.

- A bridge designer should minimize the skew introduced on the buffered clocks.
- PCI device designers should accommodate clock duty cycles which are degraded from the *PCI Local Bus Specification*.
- Motherboard designers should provide clocks to the expansion connectors that are not already at the clock duty cycle limits of the *PCI Local Bus Specification*.
- PCI expansion board designers who use bridges on their design should test for correct operation of their design over the limits of input **CLK** duty cycle.

[10] Note that **M66EN** may also be implemented as a t/s signal to qualify its ability to support 66 MHz operation based on static configuration information. The bridge may drive **M66EN** low to indicate that it cannot operate at 66 MHz (the bridge can never drive it high).

## 10.2.2. Required Signals

The secondary bus requires the same signals as the primary bus with the following exceptions:

- **IDSEL** is not required on the secondary interface, since the bridge's configuration space is not accessible from the secondary bus.
- If the bridge supports an internal secondary bus arbiter (recommended), a **REQ#** output pin and a **GNT#** input pin are not required. **REQ#** and **GNT#** are required if the bridge supports an external secondary bus arbiter.
- **RST#** is an output on the secondary bus.
- **LOCK#** is an output on the secondary bus.

Signal type:

| | |
|---|---|
| s/t/s | **FRAME#**, **IRDY#**, **TRDY#**, **STOP#**, **DEVSEL#**, **PERR#, LOCK#**[11] |
| t/s | **AD[31::00]**, **PAR**, **C/BE[3::0]#**, **REQ#** |
| Input | **CLK**, **GNT#** |
| Output | **RST#** |
| o/d | **SERR#** |

## 10.2.3. Optional Signals

The secondary interface of the bridge may optionally support the 64-bit extensions and 66 MHz operation to provide more functionality or performance as needed. The bridge may provide other PCI signal pins (such as interrupts, power management, and JTAG) or implementation-specific pins, but those listed below are expected to be the most common. Note that **REQ[*n*]#** and **GNT[*n*]#** are provided if the bridge includes a secondary bus arbiter where ***n*** signifies the number of request and grant signal pairs supported by the bridge.

Signal type:

| | |
|---|---|
| s/t/s | **REQ64#**, **ACK64#** |
| t/s | **AD[63::32]**, **PAR64**, **C/BE[7::4]#**, **GNT[*n*]#** |
| Input | **M66EN**[12], **REQ[*n*]#** |

[11] The *PCI Local Bus Specification* constrains the use of **LOCK#** to downstream transactions by bridges to avoid deadlock conditions. However, earlier versions of the *PCI Local Bus Specification* allowed use by any bus master. Therefore, secondary **LOCK#** must be implemented as a s/t/s signal to avoid potential bus conflicts.

[12] Note that **M66EN** may also be implemented as a t/s signal to qualify its ability to support 66 MHz operation based on static configuration information. The bridge may drive **M66EN** low to indicate that it cannot operate at 66 MHz (the bridge can never drive it high).

# Chapter 11
# Initialization Requirements

## 11.1. Reset Behavior

### 11.1.1. Secondary Reset Signal

The secondary reset signal, **RST#**, is a logical OR of the primary interface **RST#** signal and the state of the Secondary Bus Reset bit of the Bridge Control register (see Section 3.2.5.17.). The secondary interface **RST#** signal is asynchronous with respect to the secondary interface **CLK** signal. A small finite delay is permitted from a change of the state of the primary **RST#** signal or Secondary Bus Reset bit until a state change of the secondary **RST#** signal to allow for propagation delays. When the primary **RST#** signal is set, or the Secondary Bus Reset bit in the Bridge Control register is set, the secondary **RST#** signal must be asserted without any clocked state logic. Clocked state logic is allowed on the deassertion of the secondary **RST#** signal.

### 11.1.2. Bus Parking During Reset

All bridges are required to drive the secondary bus **AD[31::00]**, **C/BE#[3::0]**, and **PAR** signals to a logic low level (zero) when the secondary interfaces **RST#** is asserted. This requirement is independent of the location of the secondary bus arbiter (internal or external to the bridge). A small finite delay (a few clocks) is permitted from the assertion of the secondary **RST#** signal until the bridge drives the secondary signals to zero. This delay is intended to allow time for the synchronization of the reset event (such as the assertion of the primary interface **RST#** signal) which may be necessary for internal state machines to insure that **AD[31::00]**, **C/BE#[3::0]**, and **PAR** will be driven to zero. During the time interval (if any) from the assertion of the secondary interface **RST#** signal and the parking of the **AD[31::00]**, **C/BE#[3::0]**, and **PAR** signals to zero, the bridge must tri-state these same signals.

## 11.2. System Initialization

When bridges are present in a system, the BIOS is required to provide the following functions during the initialization process (each will be discussed in the following sections):

- Assignment of PCI bus numbers
- Allocation of address spaces (Prefetchable Memory, Memory Mapped I/O, I/O)
- Writing the IRQ number into each device
- Initializing the PCI display subsystem

### 11.2.1. Assigning Bus Numbers

The BIOS must assign PCI bus numbers to each bridge in the system. In what order they are assigned and when the assignments are made is not specified. All buses located behind a bridge must reside between the Secondary Bus Number and the Subordinate Bus Number (inclusive).

### 11.2.2. Allocating Address Spaces

When the BIOS finds a bridge, it must map all devices that reside below the bridge into one of either the I/O, the memory mapped I/O, or prefetchable memory address ranges supported by the bridge. The I/O address range (see Section 4.2.) has a minimum granularity of 4 KB (aligned) and a maximum of 64K. The I/O range may be restricted to the first 64 KB of the PCI I/O address space (0000 0000h to 0000 FFFFh) if only 16 bit I/O addressing is supported. If 32-bit I/O addressing is supported, then the I/O address range is permitted to reside anywhere in the 4 GB PCI I/O address space (the 4 KB granularity and size restrictions still apply). The memory mapped I/O range (see Section 4.3.) can reside anywhere in the low 4 GB of the memory space with a granularity of 1 MB (aligned). The prefetchable memory range (see Section 4.4.) can reside in a 32 or 64-bit address space with a minimum of 1 MB (alignment).

Since bridges have only one range for each address space type (I/O, memory mapped I/O, or prefetchable memory), the BIOS must group all devices that use the same type of space into a single range. This implies that when a bridge has multiple bridges and/or devices behind it, the BIOS must be able to group them into a single range per address space type.

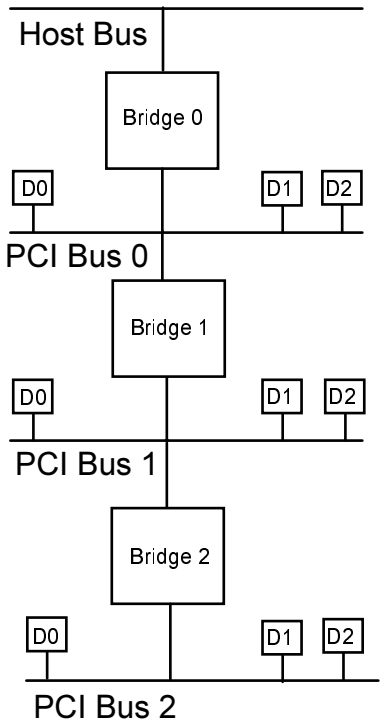



**Figure 11-1: Bus Numbering Example**

For example, in Figure 11-1, device 0 on bus 1 and device 2 on bus 2 both require I/O space. (All other devices do not require I/O space.) For the bridge (bridge 1) that resides between buses 0 and 1 to handle the decode of I/O space, the two requests for I/O space need to be mapped such that bridge 1 can decode all I/O transactions with a single range decode and forward them down. In this example, which is illustrated in Figure 11-2, device 0 on bus 1 would be assigned an I/O range (by programming its I/O BAR) and device 2 on bus 2 would be assigned a different I/O range (by programming its I/O BAR). However, for the bridge to decode them using a base and limit register, these two ranges need to be adjacent to each other. Bridge 1's I/O Base register is programmed with the value of device 0's I/O BAR register while bridge 1's I/O Limit register is programmed with the value of device 8's I/O BAR register. Bridge 0 (the host bridge) would be programmed the same as bridge 1.

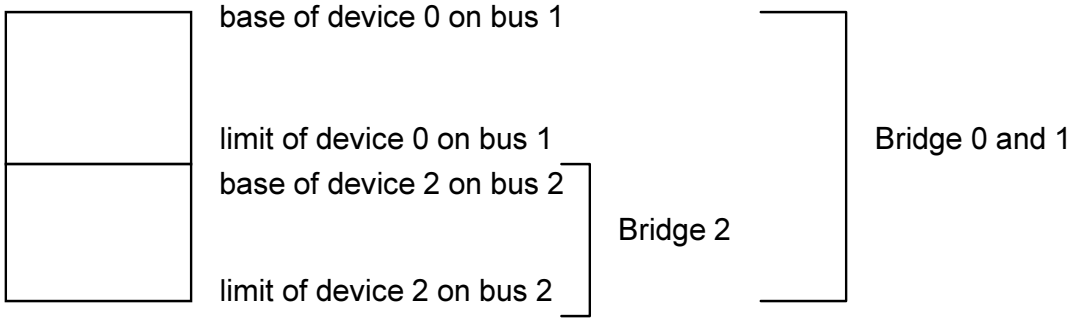



**Figure 11-2: Example of Address Range Coalescing**

The same methodology is required for the memory mapped I/O space and prefetchable memory space. The only difference is the number of devices that need to be assigned space and the size of the space for each device. To minimize the wasted space, the largest requests should be allocated first then filled in with smaller requests in the same contiguous range.

To summarize, all devices behind a given bridge must be allocated I/O, memory mapped I/O, or prefetchable memory space in such a way that the bridge can implement a single address range for each type.

## 11.2.3. Writing IRQ Numbers into Interrupt Line Register(s)

The BIOS must write the IRQ (or vector number) into each device's Interrupt Line register when an Interrupt Pin is supported. For motherboard devices, the BIOS knows how a device's interrupt pin is routed to the interrupt controller. For devices that can be added into the system by a PCI connector (expansion boards), a pre-defined routing is required for devices that reside behind a bridge. This routing is described in Section 9.1.. For expansion boards with bridges, the BIOS uses the device number and bus number to determine which IRQ line of the interrupt controller each device **INTx#** pin is connected to.

Table 11-1 is an example of a routing where:

- device 2 is on bus 2
- bridge 2 is device 3 on bus 1
- bridge 1 is device 4 on bus 0

As shown in Table 11-1, using the routing specified by Section 9.1., device 2 on bus 2 (**INTA#** pin of the device) is connected to **INTC#** of bus 2. **INTC#** on bus 2 is connected to **INTB#** on bus 0.

**Table 11-1: Interrupt Routing Example**

| Device # | Bus # | INTx# of Device | INTx# on Bus |
|---|---|---|---|
| 2 | 2 | **INTA#** | **INTC#** |
| 3 | 1 | **INTC#** | **INTB#** |
| 4 | 0 | **INTB#** | **INTB#** |

## 11.3. PCI Display Subsystem Initialization

### 11.3.1. Initial Conditions

As specified in the *PCI Local Bus Specification* and in this document, the configuration bits controlling a bridge's or display device's response to VGA accesses is hardware initialized at power-up to the following states:

- PCI-to-PCI bridges (PPBs) power-on to ignore all VGA accesses

  VGAEnable --> 0

  VGASnoopEnable --> 0

- Non-VGA compatible devices (GFXs) power-on to snoop VGA palette writes

  VGAPaletteSnoop --> 1

- VGA compatible devices (VGAs) power-on to "not snoop" VGA palette writes

  VGAPaletteSnoop --> 0

### 11.3.2. Initialization Algorithm

1. Identify boot VGA device, VGABoot. Search ISA/EISA first, then search PCI bus hierarchy top to bottom (i.e., starting at PCI Bus 0). The first VGA device encountered is the VGABoot device. If VGABoot is found on ISA/EISA, display initialization is complete. If found on PCI, save the PCI bus number, VGABusNum (where VGABoot resides), and continue with steps 2-6.
2. Enable VGABoot's response to all VGA I/O and memory spaces by setting I/O Space and Memory Space enables in the device's configuration Command register.
3. Starting at PCI bus VGABusNum, traverse the bus hierarchy up towards PCI bus 0. For each bridge passed, set that bridge's *VGAEnable* configuration bit.
4. Starting at PCI bus number VGABusNum, search the PCI bus hierarchy under bus PCIBusNum (i.e., all buses secondary to PCIBusNum) bottoms-up scanning for GFXs. Once the first GFX device is found, flag that a GFX was found and discontinue all downstream bus searches. Set the I/O Enable[13] and VGAPaletteSnoop in that GFX's configuration Command register. If no GFX is found downstream, then display initialization is complete. Otherwise, continue with steps 5-6.

[13] If the device has a valid I/O address range, its Base Address Register must be programmed. The value could be temporary or permanent depending on the initialization algorithm. Some value is required to keep the device from responding to I/O addresses.

5. Traverse back up the bus to VGABusNum. At each bridge passed, set that bridge's VGASnoopEnable bit.
6. Set VGAPaletteSnoop in VGABoot's configuration Command register. If not, clear the bit.

## 11.3.3. Algorithm Pseudo-code

```
/* This function configures all PCI-to-PCI bridges, VGAs and GFXs
participating in the boot procedure.  Once executed, POST code
may display to all devices via VGA access. */
DisplayInit()
{
      IdentifyBootVGA(VGABoot,VGABusNum,BootVGAonPCI);
      EnableVGADevice(VGABoot);
      if BootVGAonPCI
      {
            for (all PCI-to-PCI bridges upstream of VGABusNum)
                  PPB.VGAEnable = 1;
            GFXScanR(VGABusNum);
      }
      if GFXFound
            VGABoot.VGAPaletteSnoop = 1;
}

/*This function implements a recursive traversal algorithm that
scans the bus tree below BusNum bottoms-up for the first GFX
device.  Once it finds that device, it returns to bus where the
VGA device resides, configuring the PCI-to-PCI bridges and GFXs
along the way.  It returns GFXFound if any GFX was found under
BusNum.*/
GFXScanR(BusNum)
{
      for ((each PCI-to-PCI bridge on BusNum) && !GFXFound)
      {
            GFXScanR(PPB.SecondaryBusNumber);
            if GFXFound
                  PPB.VGASnoopEnable = 1;
      }
      for ((each GFX on BusNum) && !GFXFound)
      {
            GFX.VGAPaletteSnoop = 0;
            GFXFound = 1;
      }
}
```

# Chapter 12
# VGA Support

## 12.1. VGA Support

There are two issues related to the support of VGA compatible devices in systems with bridges: ISA compatible addressing and palette snooping. To support a VGA device downstream of a bridge, the bridge must have the capability to be configured to recognize the ISA compatible addresses used by VGA devices. A bridge must also support configurations where a graphics device downstream of the bridge needs to snoop VGA palette accesses.

### 12.1.1. VGA Compatible Addressing

The VGA Enable bit in the Bridge Control register (see Section 3.2.5.17.) is used to control response by the bridge to both the VGA frame buffer addresses and to the VGA register addresses. When a VGA compatible device is located downstream of a PCI to PCI bridge, the VGA Enable bit must be set. When set, the bridge will positively decode and forward memory accesses to VGA frame buffer addresses and I/O accesses to VGA registers from the primary to secondary interface and block forwarding of these same accesses from the secondary to primary interface (see Section 4.5.1.).

VGA memory addresses:

0A 0000h through 0B FFFFh

VGA I/O addresses (including ISA aliases address - **AD[15::10]** are not decoded):

**AD[9::0]** = 3B0h through 3BBh and 3C0h through 3DFh

The bridge does not decode or forward VGA BIOS memory addresses when the VGA Enable bit is set. ROM code provided by PCI compatible devices may be mapped to any address in PCI memory address space via the Expansion ROM Base Address register in the device's configuration header and must be copied to system memory before execution.

## 12.1.2. VGA Snooping

Snooping is essentially a broadcast mechanism and suffers from the lack of flow control. In a PCI system, the presence of hierarchical buses and subtractive decoding agents further complicate the issues associated with the lack of flow control. Hierarchical bus structures are easily created in PCI systems through the application of bridges. In many PCI systems, a subtractive decoding expansion bus bridge will typically exist for ISA or EISA. The solution for VGA palette snooping requires restrictions on configurations and appropriate support in VGA compatible devices, PCI-to-PCI bridges, and graphics devices that do not have VGA compatibility. Expansion bridges that use subtractive decoding for VGA palette addresses do not require any special support for VGA palette snooping.

There are basically four types of devices that must be considered when specifying a VGA palette snooping mechanism for PCI:

- VGA compatible graphics devices;
- subtractive decoding expansion bus bridges;
- PCI-to-PCI bridges; and
- graphics devices that do not have VGA compatibility.

Each of these devices may have to participate in VGA palette accesses. The behavior of each type of device must be selected based on the configuration of the system. The different behaviors of each device type are described below.

The VGA palette addresses are as follows (inclusive of ISA aliases - **AD[15::10]** are not decoded):

**AD[9::0]** = 3C6h, 3C8h, and 3C9h

### 12.1.2.1. VGA-compatible Graphics Devices

The VGA palette snoop mechanisms for a VGA compatible device is described in the *PCI Local Bus Specification*. VGAs must support two modes of VGA palette access:

- positively decode palette reads and writes, or
- positively decode palette reads and snoop palette writes when another agent positively decodes the palette write

The VGA palette access method is controlled by the VGA Palette Snoop bit in the configuration Command register as specified by the *PCI Local Bus Specification*. When the bit is set, the device must not respond to palette write accesses but must still respond to palette read accesses.

Note that any VGA device that interfaces to PCI directly can also be disabled from responding to VGA compatible addresses. This can be done by the Memory Space and I/O Space enable bits in the device's configuration Command register. An expansion bridge that uses subtractive decoding will not forward a VGA palette access when the transaction is positively decoded by another PCI agent.

## 12.1.2.2. Non-VGA-compatible Graphics Devices

Non-VGA compatible graphics devices must snoop VGA palette writes to provide VGA support via the VGA feature connector. Non-VGA compatible graphics devices must also support two modes of VGA palette access:

- positively decode palette writes
  if a non-VGA compatible device is the only device responding to palette writes on a given bus in the hierarchy
  at least one agent on a given PCI bus must claim the palette access
- snoop palette writes (absorb write without flow control)
  if there is another agent that positively decodes the palette write on the same bus segment

A non-VGA compatible graphics device can be identified by the Class Code and Sub-Class Code fields in the device's configuration space header (class code = 03h and sub-class code = 80h). This type of graphics device should implement the VGA Palette Snoop bit in the Command register but interpret the bit in the following way:

- VGA Palette Snoop bit is set (default state)
  when set the device must snoop palette writes
  the device does not claim the cycle by asserting **DEVSEL#**
  palette reads are ignored
- VGA Palette Snoop bit is cleared
  when clear the device must claim palette writes by asserting **DEVSEL#**
  palette reads are ignored

Note the device may provide an implementation specific method to allow the driver to configure it to ignore palette writes. This may be done when the VGA Snoop bit is clear and the device driver knows that the device does not need to provide VGA compatibility via palette snooping (i.e., is not a boot device).

## 12.1.2.3. PCI-to-PCI Bridges

A PCI-to-PCI bridge must support three modes of VGA palette access:

- ignore palette accesses
  when there are no graphics agents downstream that need to snoop or respond to VGA palette access cycles
- positively decode and forward palette writes
  when there are graphics agents downstream of the bridge that need to respond to or snoop palette writes
- positively decode and forward palette reads and writes
  when VGA compatible graphics agents that are downstream of the bridge are being used

The VGA Enable and VGA Snoop Enable bits in the Bridge Control register (see Section 3.2.5.17.) select the bridge's response to palette accesses in the following way.

| VGA Enable Bit | VGA Snoop Enable Bit | Bridge Response to Palette Accesses |
|---|---|---|
| 0 | 0 | Ignore all palette accesses |
| 0 | 1 | Positively decode palette writes (ignore reads) |
| 1 | x | Positively decode palette reads and writes |

### 12.1.2.4. Subtractive Decoding Bridges

Subtractive decoding bridges need no special support for VGA palette snooping. If another PCI agent positively decodes a palette access, that access will not be forwarded to the downstream interface of the subtractive bridge.

## 12.2. VGA Configuration Restrictions

The following configuration restrictions must be satisfied for the VGA palette snooping mechanisms to function correctly.

- The graphics agent that responds to palette reads and the graphics agent that responds to or snoops palette writes must be located along the same path in the PCI bus hierarchy (see examples of illegal configurations in Sections 12.3.12., 12.3.11., and 12.3.14.).
- When a PCI-to-PCI bridge is configured to respond to palette accesses (writes or reads and writes) a subtractive decoding agent upstream of the bridge will not be able to respond to or snoop palette accesses.

  The subtractive agent will not respond to the access since it was claimed by an agent on the PCI (i.e., the PCI-to-PCI bridge) bus.

  For example, a VGA device located on ISA/EISA cannot be used in conjunction with a VGA compatible or VGA snooping graphics device that is located behind a PCI-to-PCI bridge.
- When a PCI-to-PCI bridge is configured to positively decode palette writes (and not palette reads), a graphics device upstream of the bridge must be configured to respond to palette reads and to absorb (snoop) palette writes:

  The bridge positively decodes the palette write for a downstream device.

  Note that a device downstream of the bridge must be configured to respond to the palette writes (i.e., assert **DEVSEL#** to prevent the cycle from terminating with a master-abort).
- When a PCI-to-PCI bridge is configured to positively decode palette writes and palette reads, a graphics device upstream of the bridge must be configured to absorb (snoop) palette writes:

  The bridge positively decodes palette reads and writes for a downstream device (i.e., a

VGA compatible device).
Note that a device downstream of the bridge must be configured to respond to the palette writes and palette reads (i.e., assert **DEVSEL#** to prevent the cycle from terminating with a master-abort).

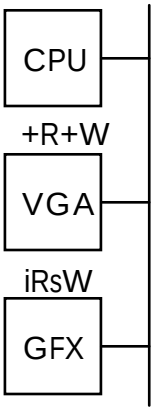
## 12.3. VGA Palette Snooping Configuration Examples

The configuration rules for VGA Palette Snooping are more easily understood by example.

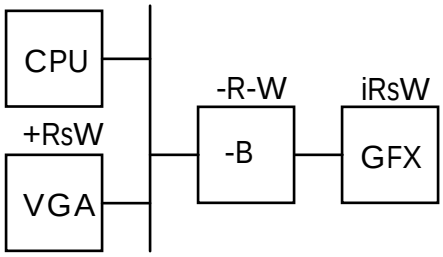
The decoding behavior for palette snooping is noted in the examples using the format xRyW where x and y describe the palette address decoding behavior of the device during reads (R) and writes (W) respectively. The symbols for x and y may be as follows:

| Symbol | Palette Address Decoding Method |
|---|---|
| - | subtractive decoding |
| + | positive decoding |
| i | ignore access |
| s | snoop access |

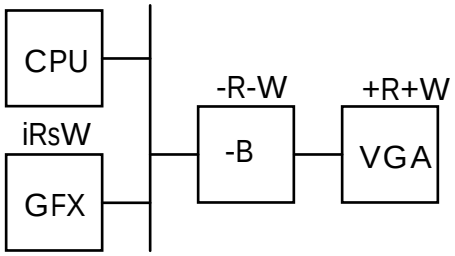
### 12.3.1. VGA and GFX on PCI Bus 0



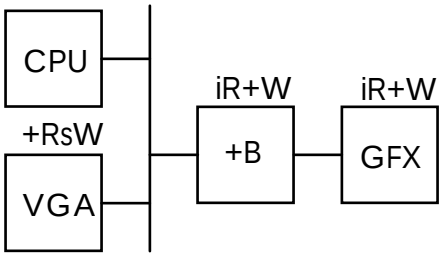
### 12.3.2. GFX Downstream of Subtractive Bridge



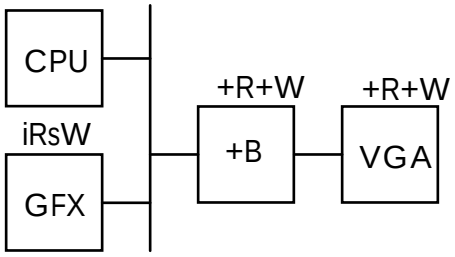

### 12.3.3. VGA Downstream of Subtractive Bridge



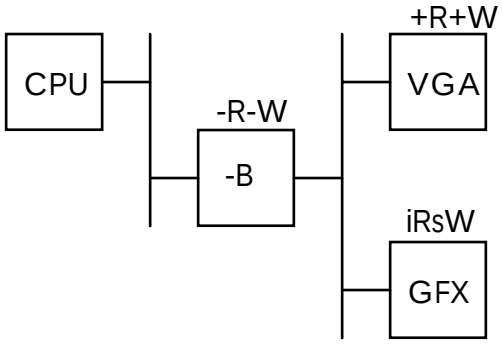
### 12.3.4. GFX Downstream of Positive Bridge



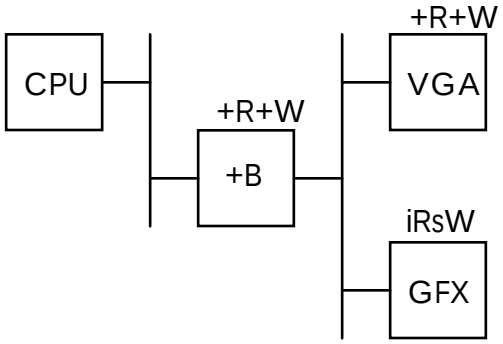

### 12.3.5. VGA Downstream of Positive Bridge



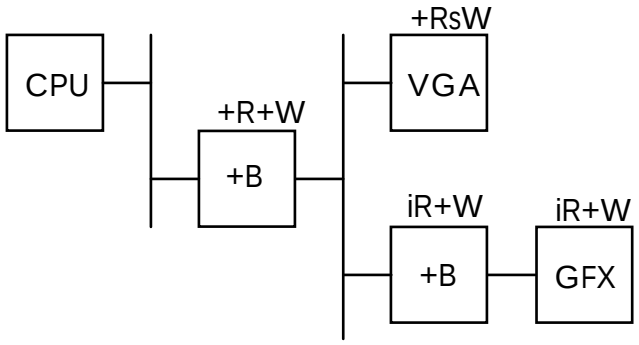
## 12.3.6. VGA and GFX Downstream of Subtractive Bridge



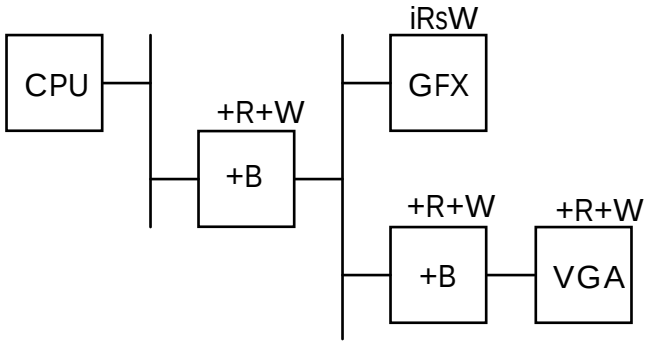

## 12.3.7. VGA and GFX Downstream of Positive Bridge



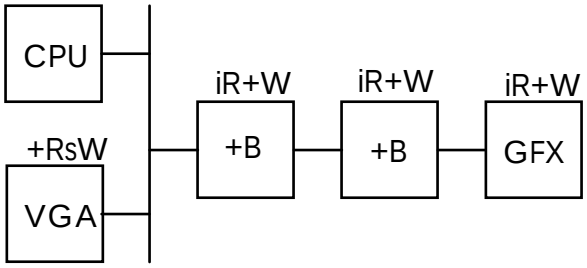
## 12.3.8. GFX Downstream of VGA on Same Path



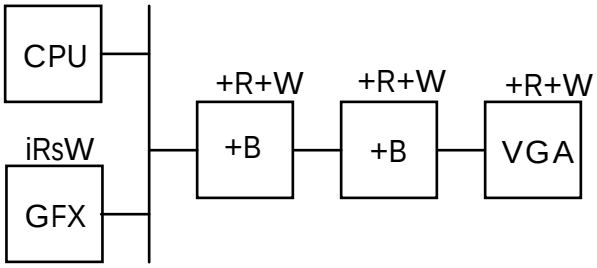

## 12.3.9. VGA Downstream of GFX on Same Path



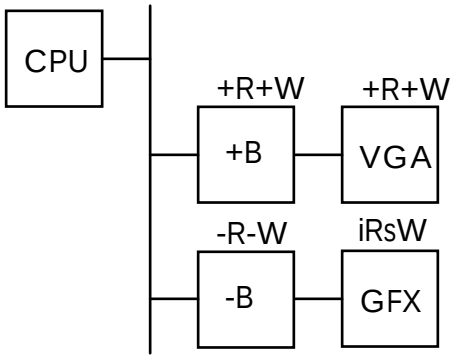
## 12.3.10. GFX Far Downstream of VGA on Same Path



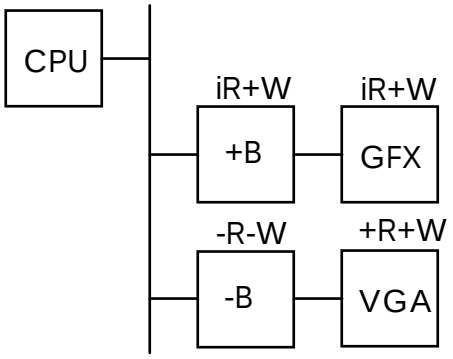

## 12.3.11. VGA Far Downstream of GFX on Same Path



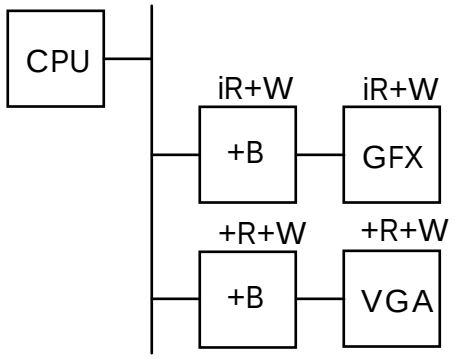
## 12.3.12. Illegal - Write Never Gets to GFX




## 12.3.13. Illegal - Write Never Gets to VGA



## 12.3.14. Illegal - Two Devices Respond to Writes



# Chapter 13
# Slot Numbering

## 13.1. Introduction

PCI system software uniquely addresses every PCI device with a bus number and a device number. These two numbers are used by system hardware to deliver Configuration transactions to the proper bus and enable (assert **IDSEL** to) the proper device.

In addition to this unique logical identification, many situations require the unique physical identification of each device. Physical identification of a particular expansion board is often simple if there is only one board of each type in the system. But the situation becomes more difficult if the system contains multiple expansion boards of the same type. The following situations illustrate the need for the user physically to identify a particular expansion board, when multiple identical expansion boards are installed in a system:

1. When plugging in an external cable, the user must identify the appropriate expansion board.
2. When configuring items such as the operating system, device drivers, and protocol stacks, the user must identify the device to the software. For example, when configuring network controllers, the user must typically specify a controller (identified by slot number) and then assign a network address and the protocols to use with that controller.
3. If an expansion board fails in a system, software such as diagnostic tools must have a way to identify to the user which expansion board has failed so that the user can physically replace it.
4. Hot-plug operations are required by the *PCI Hot-Plug Specification* to use physical slot numbers in the user interface.

PCI bus numbers throughout the system can change with the introduction or removal of one PCI-to-PCI bridge anywhere in the system, so bus and device numbers are not suitable for the user to physically identify the device. The slot number is the constant physical property by which the user identifies an expansion board. Software must translate between this constant physical property (slot number) used by the human user and the logical identifier (bus and device number) used by the system software.

Systems containing PCI slots typically partition the slots physically into two kinds of groups. Algorithms for translating between logical and physical slot identifications vary depending upon in which kind of group the slot is located. The group associated most closely with the CPU that

initializes the PCI Configuration Space is called the *main chassis*. A group of PCI slots that can be connected to and disconnected from the main chassis is said to reside in a PCI *expansion chassis*. At the system vendor's option, an expansion chassis can be tightly coupled physically to the main chassis or can be separated from the main chassis by a great distance. In general, if a chassis containing PCI slots can be connected and disconnected from a main chassis, that chassis is considered an expansion chassis and must use the slot numbering mechanisms described here for expansion chassis. If an expansion chassis does not use the slot numbering mechanisms described here, the system vendor must provide a proprietary mechanism for translating between logical and physical slot identifiers.

For slots in the main chassis the translation between logical bus and device numbers and physical slot numbers is made possible by a slot number field in the IRQ routing table in system ROMs that follow the *PCI BIOS Specification*[14]. For PCI expansion chassis the same translation is made possible by two registers, Chassis Number and Slot Number, in certain PCI-to-PCI bridges. The registers are required only in some bridges because they are unnecessary for bridges on expansion boards or in the main chassis. They are required in bridges that provide additional expansion slots in expansion chassis.

PCI expansion chassis are often built with multiple PCI-to-PCI bridges arranged in a hierarchical fashion. Three examples are shown in Figures 13-3 through 13-5. The slot numbering algorithm presented later in this chapter calculates the slot number of a device by traversing the bus hierarchy in a predetermined order. Each time the algorithm encounters the beginning of a new chassis, it remembers the chassis number and begins accumulating the number of expansion slots until it reaches either the device it is trying to identify or the beginning of another chassis. Device numbers for expansion slots, bridges, and other devices must be assigned in a specific order to enable the algorithm properly to determine the chassis and slot number.

The remainder of this chapter focuses on three areas of requirements for slot numbering in expansion chassis:

1. Expansion chassis device numbering requirements. System software assumes PCI device numbers are assigned in a particular order in expansion chassis.
2. The slot numbering registers in bridges used to create expansion slots.
3. The software algorithm that uses the expansion chassis device numbering assumptions and the contents of the slot numbering registers to translate between logical and physical device identifications for devices located in PCI slots in expansion chassis.

## 13.2. Device Number and Slot Number Assignment Rules

System software assumes that PCI device numbers (that is, the number that appears in **AD[15::11]** of a Type 1 Configuration transaction, and is converted to **AD[31::16]** of a Type 0 Configuration transaction by the bridge) are assigned to devices on the secondary bus in a particular order in PCI expansion chassis. These assumptions are combined with the contents of the Slot Number and Chassis Number registers to enable the software algorithm shown in Figure 13-6 to translate between bus/device number and chassis/slot number.

[14] *PCI BIOS Specification*, Revision 2.1, PCI Special Interest Group, August 26, 1994.

1. Slot numbers within a single expansion chassis start at 1 and increment sequentially.
2. The PCI device numbers for each expansion slot on the same bus segment start at 1 and increment sequentially. Embedded devices (including bridges to other buses or slots) must be assigned device numbers higher than the last slot on that bus. Furthermore, embedded bridges must be assigned device numbers highrer than all other embedded devices. Device number 0 is not used for any device (that is, **AD16** from the PCI-to-PCI bridge is not connected to any **IDSEL** pin).
3. If a single expansion chassis includes multiple bridges that support expansion slots on their secondary buses, the slot numbers for each bridge must increment sequentially (no repeated or missing slot numbers) from one bridge to the next. The bridges must be arranged so they are discovered by the slot numbering algorithm (described later) in the same order as their respective slot numbers. In other words, such bridges must be arranged as follows:
    a. If they share the same primary bus, the bridge device numbers must be assigned in the same order as the slot numbers.
    b. If they are arranged hierarchically, a superior bridge (i.e., a bridge closer to the CPU) must have lower slot numbers than all its subordinate bridges (i.e., bridges whose primary bus numbers fall between the secondary and subordinate bus numbers of the superior bridge, inclusive).

Any bridge anywhere in the hierarchy can be implemented with no slots directly on its secondary bus. Bridges embedded on a expansion boards commonly have no slots behind them. Figure 13-4 shows an example of bridge in an expansion chassis with no slots on its secondary bus.

All the devices and functions on a single expansion board use the same chassis number and slot number. (An adapter card such as a multi-headed NIC or SCSI controller behind an embedded bridge reports the same chassis number and slot number for all devices on that card.)

Specialized bridges (bridges other than general-purpose PCI-to-PCI bridges) can also be used to connect to expansion chassis. Such bridges must use a Type 1 configuration header and include the slot numbering registers in their configuration space, if they are to be included in the standard slot numbering scheme of the system. Software routines that are designed to discover slot numbering registers only in bridges with Type 1 configuration headers will not discover slots behind other types of bridges. Expansion slots behind bridges that do not follow this specification require specialized software to translate between bus/device number and chassis/slot number.

It is theoretically possible to design a specialized bridge that physically connects an expansion slot in one chassis to slots in another chassis, and the bridge appears as a single logical device in the PCI bus hierarchy. For example, in Figure 13-3 the bridge at the left labeled the “source” bridge and Bridge A could be combined into a single logical device. However, such an implementation is discouraged. In Figure 13-3 if the source bridge and Bridge A are combined, the combined bridge must be marked as the first bridge in the expansion chassis, so the new chassis number can be recorded, and to indicate that the next slot is slot number 1. In that case, the source bridge would appear (to the slot numbering algorithm) to be located in the expansion chassis, even though the first part of the bridge is physically located on an expansion board in the main chassis. The recommended implementation is for the source bridge always to appear in the

PCI bus hierarchy as a separate logical bridge device from the first bridge in the expansion chassis.

Additional devices and functions are permitted to be integrated with bridges to expansion chassis. Devices that appear in the hierarchy directly on the secondary bus of the source bridge are reported as being embedded devices in the same chassis and slot as the source bridge. Devices that appear in the hierarchy on the secondary bus of a bridge in an expansion chassis are reported as being embedded devices in the expansion chassis. Note that a device integrated with the bridge but appearing in the PCI bus hierarchy as a separate device on the bridge's secondary bus is still required to use a device number higher than the last slot of the bridge's secondary bus.

## 13.3. The Slot Number Register

The slot numbering registers are generally optional. They are required for bridges used in the following applications:

- The first bridge in a PCI expansion chassis
- All bridges in an expansion chassis that have PCI slots on their secondary bus

One of the slot numbering registers, the Expansion Slot register, is shown in Figure 13-1. The Expansion Slot register is initialized by hardware before the system's PCI configuration routine executes. The configuration routine uses the contents of this register to determine how many physical slots are present on the secondary bus of this bridge, and how they are connected. If the Expansion Slot register in a bridge is not implemented, the PCI configuration routine assumes that no slots exist directly on the secondary bus of this bridge.

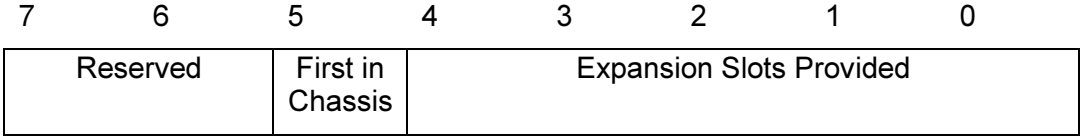



**Figure 13-1: Expansion Slot Register**

Bits 4-0, the Expansion Slots Provided field, contains the binary encoded value of the number of expansion slots that are provided directly on the secondary bus of this bridge. If no expansion slots are implemented on the secondary bus of a particular bridge, this field is initialized to 0.

The first bridge in the expansion chassis (that is, the bridge with the lowest primary bus number, or the lowest device number among bridges with the same primary bus number) will have the First in Chassis bit set to 1. Additional bridges (subordinate to the first bridge or those having higher device numbers on the same bus) have the First in Chassis bit reset to 0.

## 13.4. The Chassis Number Register

Each chassis containing PCI slots is assigned by PCI configuration software a unique chassis number between 0 and 255 inclusive. The main chassis is always chassis number 0. Chassis numbers need not be sequential.

Figure 13-2 shows the Chassis Number register. If a bridge supports the slot numbering registers, PCI configuration software will initialize the Chassis Number register in that bridge. All bridges in the same chassis are initialized with the same chassis number, even if the bridge has no expansion slots on its secondary bus (e.g., a bridge on an expansion board).

Run-time software will read the chassis number from this register when translating between bus and device number and chassis and slot number.

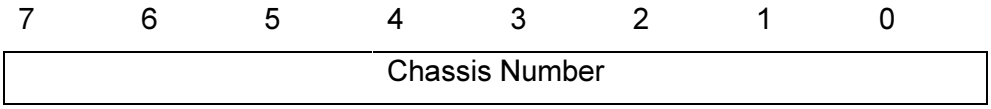



**Figure 13-2: Chassis Number Register**

The designer of the PCI-to-PCI bridge that includes the Chassis Number register has two options for initialization:

1. Initialized to 0 at reset.
2. Non-volatile, not affected by reset.

If the Chassis Number register is initialized to 0 at reset, system initialization software will assign a chassis number each time the bridge is reset. This option requires the least amount of hardware of the two initialization options. However, if a chassis is installed or removed from the system, or if connections to expansion chassis are rearranged in the main chassis, all chassis numbers throughout the system might change from one reset event to the next.

If the Chassis Number register is non-volatile and has previously been initialized, system initialization software will discover a non-zero number in the Chassis Number register and not modify it (unless it is discovered to conflict with another chassis number; e.g., if a new chassis was added). This option provides the most advantages to the user, since the chassis numbers never change unless a new expansion chassis is added that uses the same number as an existing one.

## 13.5. A Slot Numbering Example

The diagrams shown in Figure 13-3 through Figure 13-5 contain all the elements that affect the numbering of PCI expansion slots. In both Figure 13-3 and Figure 13-4 a single PCI expansion chassis containing a hierarchy of bridges is connected to the main chassis via the PCI-to-PCI bridge on the left side (the arrow indicates the connection to the main chassis). Figure 13-5 represents a system in which multiple expansion chassis are connected to the main chassis through a single device in the main chassis. All bridges in the expansion chassis in this example are connected as peers in the PCI bus hierarchy.

Above each expansion slot in the figures is the slot number that is physically labeled on the slot. The other numbers shown are the PCI device numbers that are assigned to embedded devices, or would be assigned to devices installed in slots.

In Figure 13-3, PCI-to-PCI Bridge A is the first bridge in the expansion chassis, so its First in Chassis bit is set. This bridge supports slots directly on its secondary bus (Bus 1), so slot 1 is

there. Slots must be numbered sequentially (starting at 1) and device numbers are equal to slot numbers. Also on Bus 1 is an embedded PCI device. It is assigned device number 5, since it is permitted to be any number higher than the last slot on Bus 1.

Also on Bus 1 are PCI-to-PCI Bridges B and C, each with three expansion slots on their secondary interfaces. Since both Bridges B and C are subordinate to Bridge A, the First in Chassis bits for both Bridges B and C are cleared. Both are initialized with the same chassis number as Bridge A. Furthermore, since Bridge B sources the bus with the lower numbered slots, its device number on Bus 1 must be smaller than the device number for Bridge C. Accordingly, Bridges B and C are assigned device numbers 6 and 7 respectively. Device numbers for the slots behind the subordinate bridges are assigned sequentially starting with device number 1, even though the slot numbers continue in sequence after the slots for the previous bus segment.

Bridges D and E are examples of bridges embedded on expansion boards. Bridge D typifies a bridge that does not implement slot numbering registers, and Bridge E typifies a bridge that does. Since Bridge E has no slots behind it, its Expansion Slots Provided field is initialized to 0 and its First in Chassis bit is cleared by hardware. PCI configuration software initializes the chassis number to that of the chassis in which the expansion board is installed.

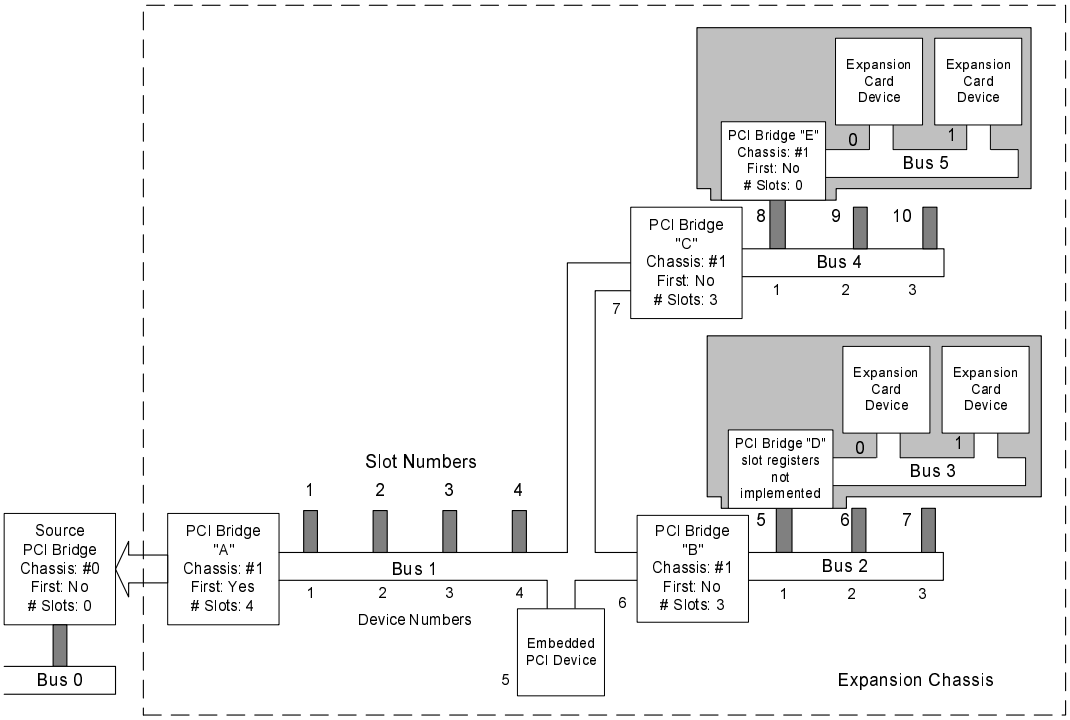



**Figure 13-3: Example PCI Expansion Chassis with Slots on First-in-Chassis Bridge**

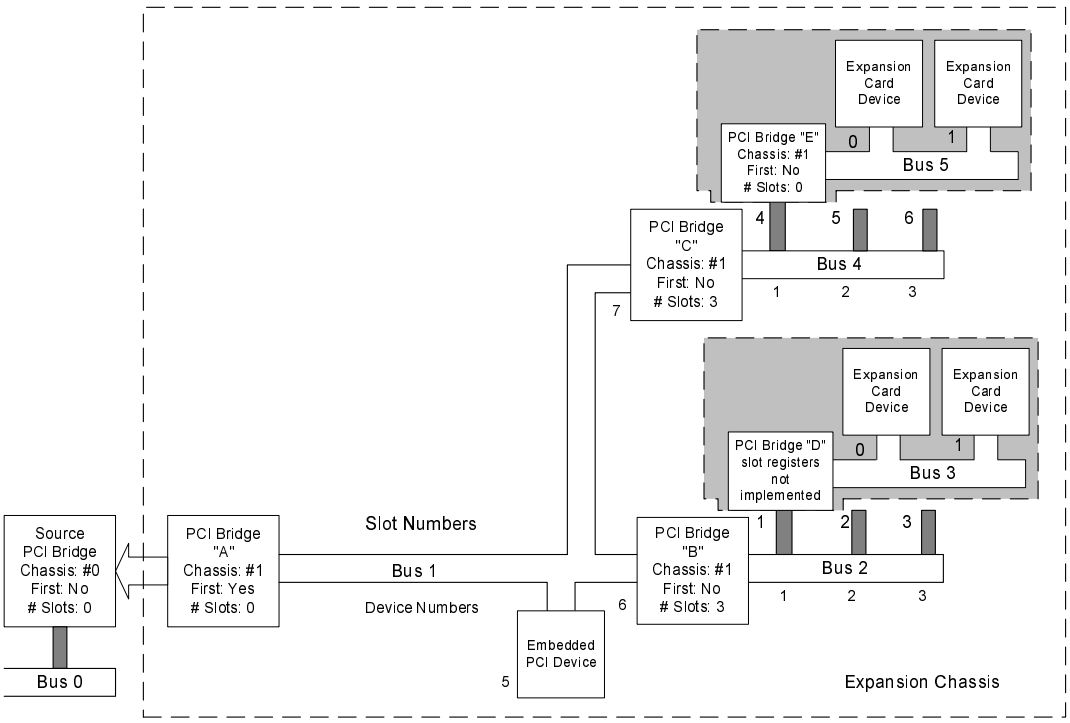



**Figure 13-4: Example PCI Expansion Chassis Without Slots on First-in-Chassis Bridge**

In Figure 13-4, the first bridge in the expansion chassis has no slots directly on its secondary bus. As in Figure 13-3, the expansion chassis begins with Bridge A, but Bridge A supports only an embedded device (device number 5), not any expansion slots. Bridge B supports slot 1 since it is the bridge with the lowest device number on Bus 1. All the bridges in the expansion chassis are initialized with the same chassis number. Device numbers for the slots behind the Bridge C are assigned sequentially starting with device number 1, even though the slot numbers continue in sequence after the slots for the previous bus segment.

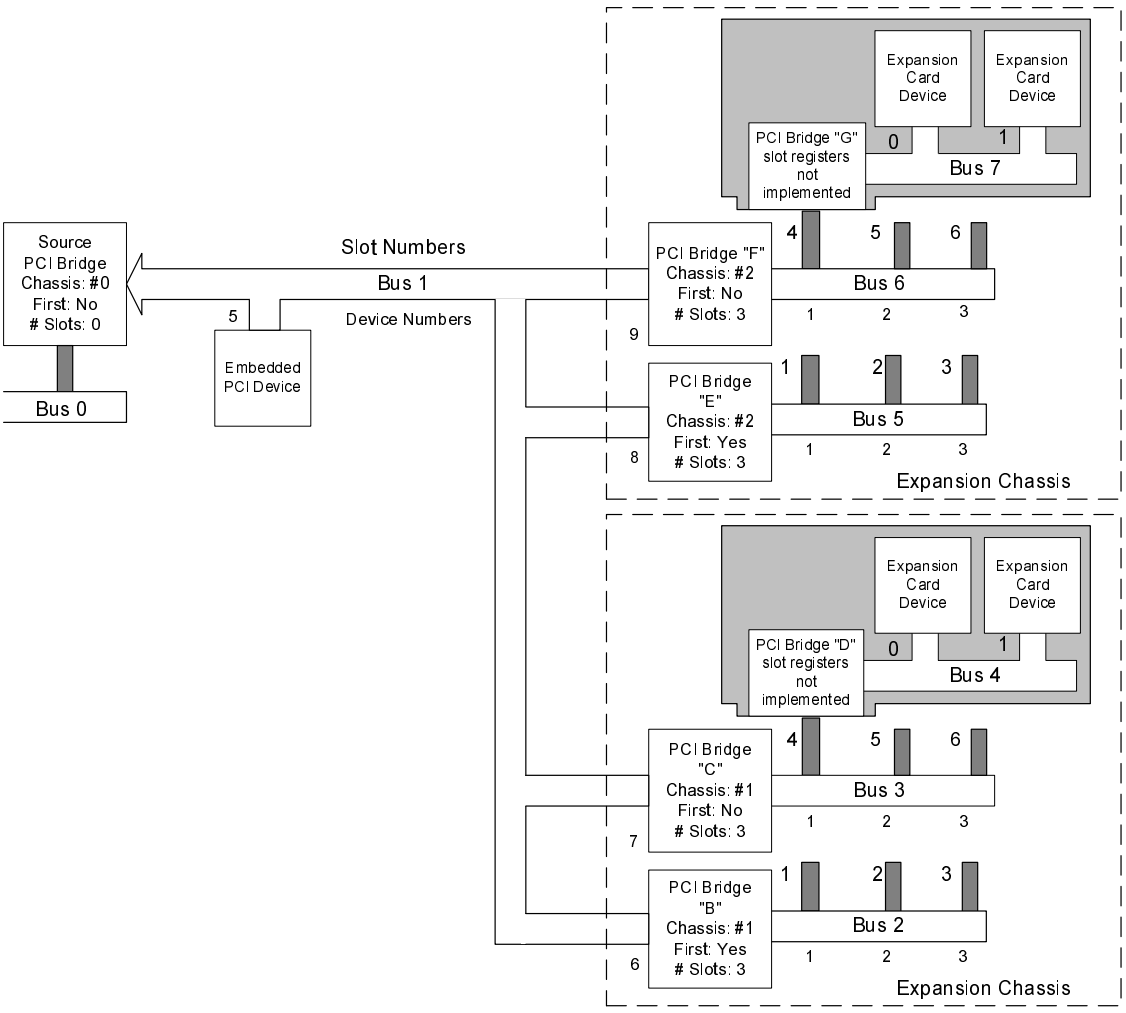



**Figure 13-5: Example PCI Expansion Chassis With Peer Bridges**

Figure 13-5 illustrates another arrangement for bridges to expansion chassis. Figure 13-5 shows multiple bridges in a single expansion chassis connected as peers on the bus to the main chassis. In this example the embedded device on Bus 1 is in the main chassis and reports the same slot number as the source PCI bridge. (The slot numbering algorithm does not allow for a device other than a bridge on Bus 1 to report a location anywhere other than the location of the source PCI bridge.)

In Figure 13-5, Bridge B is the bridge with the lowest device number on Bus 1, so it is the first bridge discovered by the algorithm in Chassis #1. Its First in Chassis bit is set and it supports slot 1. Bridge C has the next higher device number, so its slot numbers follow those of Bridge B. Bridge E begins Chassis #2, which in this example is identical to Chassis #1.

## 13.6. Run-Time Algorithm for Determining Chassis and Slot Number

The algorithm for determining the chassis and slot number of a device is shown in Figure 13-6. It starts by checking the IRQ Routing Table for information about how the main chassis is connected. If the device in question is plugged directly into a main-chassis slot, the slot number can be read directly from the IRQ Routing Table using PCI BIOS routines.

If the device in question is not plugged directly into a main-chassis slot, the algorithm must scan the PCI bus hierarchy to locate the device. For systems with multiple host bridges the algorithm uses the IRQ Routing Table to determine which host bridge sources the branch of the hierarchy supporting the bus for the device in question. The algorithm searches the PCI bus hierarchy in a predefined order, starting from the appropriate host bridge. If the algorithm finds the bus and device without ever encountering a bridge with the First in Chassis bit set, the device is located in the main chassis, and its slot number is deduced from the IRQ Routine Table.

If the algorithm finds a bridge with the First in Chassis bit set, it has found a new expansion chassis. Whenever the algorithm finds a new chassis, it picks up the chassis number from the Chassis Number register and starts counting expansion slots. The algorithm proceeds through the bus hierarchy counting slots based on the device number and the Slots Provided register in each bridge that it finds along the way. Note that even though the algorithm only reads the Chassis Number register from the bridge with the First in Chassis bit set, system initialization software is required to initialize all Chassis Number registers in the same chassis to the same value.

```
Finding a bus/device/function's chassis/slot number

Definitions:

PIRT = PCI IRQ Routing Table
FBC = First Bridge in Chassis (bit in bridge)
bridge leading to bus = bridge where Secondary Bus Bumber <= bus and
Subordinate Bus Number >= bus
done = SLOT and CHASSIS have the desired results

Temporary Variables:

NUMSLOTS = intermediate storage
DEVICE = intermediate storage of a device/function number
BUS = bus number currently being searched
PPB = bus/device/function number of PCI to PCI (P2P) bridge being examined

Inputs:

TBUS = Bus number of target device
TDEVICE = Device number of target device

Outputs:

CHASSIS = Chassis number result (0 = Main)
SLOT = Slot number result (0 = Embedded)

Main routine

CHASSIS = 0

// Check for an entry in the PCI IRQ Routing Table
if target is in PIRT
      {
      SLOT = slot number from PIRT
      done
      }

// Start searching for the target
Set BUS to highest bus in PIRT less than TBUS

Find the P2P bridge on BUS leading to TBUS
PPB = P2P bridge leading to TBUS

SLOT = Slot number of PPB (from PIRT)

Set BUS to Secondary bus number from PPB

Label SCANCHASSIS
```

```
NUMSLOTS = Number of expansion slots from PPB (may be zero)

// Check for the beginning of a new chassis
   if PPB is FBC
      {
      CHASSIS = Chassis number from PPB
      SLOT = 0
      }

for DEVICE = device 0 to device 31 and all functions
      {
      // Increment the slot number only for device numbers that correspond to
          slots
      if DEVICE > 0 and DEVICE <= NUMSLOTS
            SLOT = SLOT + 1

      // Check for a match
      if BUS matches TBUS and DEVICE matches TDEVICE
            {
            // Embedded devices must be after slots if they aren't directly
               behind the first bridge in a chassis
            if NUMSLOTS > 0 and DEVICE > NUMSLOTS
                  SLOT = 0
            if DEVICE is bridge and DEVICE is FBC
                  {
                  CHASSIS = Chassis number from DEVICE
                  SLOT = 0
                  }
            done
            }

      // Check for a bridge that must be traversed
      if DEVICE is bridge leading to TBUS
            {
            PPB = DEVICE

            Set BUS to Secondary bus number from PPB

            goto SCANCHASSIS:
            }

      // Check for a bridge that has slots which must be counted, i.e. a
         bridge with slots behind it but TBUS is behind another bridge
      // Note that bridges in slots are not included in this
      if DEVICE > NUMSLOTS and DEVICE is bridge
            {
            // Check for a bridge which signals the start of a new chassis
            if DEVICE is FBC
                  {
                  CHASSIS = Chassis number from DEVICE
                  SLOT = 0
                  }
            SLOT = SLOT + Number of expansion slots from DEVICE
            }
      }
```

**Figure 13-6: Example Algorithm for Converting Bus/Device Number to Chassis/Slot Number**